AD-A056 534 ILLINOIS UNIV AT URBANA-CHAMPAIGN COORDINATED SCIENCE LAB F/G 9/5
DESIGN OF DIGITAL SYSTEMS USING SELF-CHECKING ALTERNATING LOGIC--ETC(U)
OCT 77 S E WOODARD DAAB07-72-C-0259
UNCLASSIFIED R-788 NL

1 OF 2
AD A056534

REPORT R-788 OCTOBER, 1977 UILU-ENG 77-

CSL COORDINATED SCIENCE LABORAT

LEVEL II

DESIGN OF DIGITAL SYSTEMS USING SELF-CHECKING ALTERNATING LOGIC

SCOTT EUGENE WOODARD

UNIVERSITY OF ILLINOIS - URBANA ILL

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) DESIGN OF DIGITAL SYSTEMS USING SELF-CHECKING ALTERNATING LOGIC	5. TYPE OF REPORT & PERIOD COVERED Technical Report
	6. PERFORMING ORG. REPORT NUMBER R-788, UILU-ENG-77-2235
7. AUTHOR(s) Scott Eugene Woodard	8. CONTRACT OR GRANT NUMBER(s) DAAB-07-72-C-0259
9. PERFORMING ORGANIZATION NAME AND ADDRESS Coordinated Science Laboratory University of Illinois at Urbana-Champaign Urbana, Illinois 61801	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Joint Services Electronics Program	12. REPORT DATE October, 1977
	13. NUMBER OF PAGES 115
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report) UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Self-Checking Alternating Logic
Combinational Logic
Sequential Logic

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This thesis presents further analysis of the application of alternating logic to the design of self-checking systems. In particular, results are presented in the areas of combinational logic, sequential logic, self-checking alternating logic modules, self-checking alternating logic checker design, and self-checking alternating logic system design.

The necessary and sufficient conditions for a self-dual combinational network to be self-checking are developed. An analytic technique for evaluating if any self-dual network is self-checking is given. A memory efficient approach

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. ABSTRACT (continued)

for the design of self-checking alternating logic sequential machines is presented. Various techniques of checker design for self-checking alternating logic are discussed. The requirements of the hardcore portion of general self-checking systems is given. Minority modules are shown to be sufficient to convert any NAND or NOR network to a self-checking alternating logic network.

UILU-ENG 77-2235

DESIGN OF DIGITAL SYSTEMS USING SELF-CHECKING ALTERNATING LOGIC

by

Scott Eugene Woodard

This work was supported in part by the Joint Services Electronics Program (U.S. Army, U.S. Navy and U.S. Air Force) under Contract DAAB-07-72-C-0259.

DESIGN OF DIGITAL SYSTEMS USING SELF-CHECKING ALTERNATING LOGIC

BY

SCOTT EUGENE WOODARD

B.S., University of Illinois, 1973
M.S., University of Illinois, 1975

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1977

Thesis Adviser: Professor Gernot Metze

Urbana, Illinois

DESIGN OF DIGITAL SYSTEMS USING

SELF-CHECKING ALTERNATING LOGIC

Scott Eugene Woodard, Ph.D.
Coordinated Science Laboratory and
Department of Electrical Engineering
University of Illinois at Urbana-Champaign, 1977

This thesis presents further analysis of the application of alternating logic to the design of self-checking systems. In particular, results are presented in the areas of combinational logic, sequential logic, self-checking alternating logic modules, self-checking alternating logic checker design, and self-checking alternating logic system design.

The necessary and sufficient conditions for a self-dual combinational network to be self-checking are developed. An analytic technique for evaluating if any self-dual network is self-checking is given. A memory efficient approach for the design of self-checking alternating logic sequential machines is presented. Various techniques of checker design for self-checking alternating logic are discussed. The requirements of the hardcore portion of general self-checking systems is given. Minority modules are shown to be sufficient to convert any NAND or NOR network to a self-checking alternating logic network.

ACKNOWLEDGMENT

The author would like to express his sincere gratitude for the guidance of his thesis advisor, Professor Gernot Metze. The assistance given by Margit Livingston and Jean Dussault in the production of the thesis is greatly appreciated.

The author dedicates his thesis to his family: parents Ralph and Betty and siblings Keith, Daryl, Mark, Paul, and Nancy.

TABLE OF CONTENTS

1. INTRODUCTION

1.1. Improving Digital Systems' Reliability

As the use of digital systems has increased, a need for systems with improved reliability has arisen. Although physical device reliability has improved remarkably, much improvement is needed in developing system level design approaches. This thesis will consider a method using time redundancy to detect system failure dynamically.

1.2. Failure Modeling

To develop a method of design to improve a system's reliability, it is necessary to determine how the system may fail. If the design method is to be valid, it must use a model of the possible failure modes which is sufficiently close to the actual physical failure modes. Some previously developed models are the single stuck-at fault model, the multiple stuck-at fault model, the unidirectional fault model [ANDE1] and the pin-fault model [KETE].

The single stuck-at fault model is valid when a high percentage of the physical failures in the system is manifested as logical failures on a single line during a limited time period. The failure may be permanent or transient, but must not affect more than one line's logical value. A second failure is presumed not to occur before the first one has been recognized. These assumptions are well accepted and have received considerable experimental and theoretical [BREU,SHED1] support. Consequently, it is appropriate that the single fault model be used in this thesis.

1.3. General Reliability Improvement Techniques

From Figure 1.1 the relationship of alternating logic to reliability improvement is apparent. Since some areas may overlap, the classification scheme is not precise. However, it does provide a perspective to the role of alternating logic.

Fault tolerance is used in systems to remain operable for a restricted time period after initial failure has occurred. Fault diagnosis is used in systems to locate the internal source of an observed system failure. Fault detection is used in systems to determine whether a failure has occurred; if failure has been detected, appropriate removal or repair of the failed module may be done.

For classification, fault detection may be considered as either static or dynamic. Static fault detection is done while the system is not performing its normal function; i.e., it requires the system be dedicated to the testing procedure while it is checked. For example, when programs are run to check whether certain system modules are operating properly, a static test is being performed. If the fault detection is accomplished while the system is performing its normal operation, then dynamic testing is being done.

The dynamic test may be done in software or hardware. To perform software dynamic fault detection, a check can be made on the intermediate results of a program to determine whether the answer is correct. For example, a program which finds the solution to the unknowns in a set of equations may be checked in software by substituting the results back into the equations and checking whether the equations are true. Hardware dynamic fault detection requires the use of redundant hardware. The signals representing the

Reliability Improvement

I. Tolerance

II. Diagnosis

III. Detection

A. Static

B. Dynamic

1. Software

2. Hardware

a. Space Encoding

b. Time Encoding: Alternating Logic

Figure 1.1. Position of alternating logic in realm of reliability improvement techniques

information are encoded in a manner requiring more signals than the minimum required to just represent the information. The number of additional signals required varies depending on the type of failures against which the systems is to be protected.

Dynamic fault detection with hardware is generally referred to as self-checking. The hardware redundancy may be done in space or in time. Space encoding involves a physical increase in the amount of hardware so that the redundant signals used to check the operation are processed at the same time as the information signals. This is the most commonly examined type of self-checking and several results are available [DUSS1,OZGU,PITT,WAKE1,WAKE2].

Alternating logic is a method of hardware fault detection using time redundancy. The principal characteristic is that the additional code signals used to check the system operation are obtained using essentially the same hardware as in an unchecked system, but with additional time required to generate the code signals. Such systems are useful when a slight surplus of time is available and it is desired to minimize the hardware costs to protect a system from undetected failures.

1.4. Previous Work in Self-Checking Alternating Logic (SCAL)

The first work relating to alternating logic was done by Bark and Kinne [BARK] at Raytheon in 1956. To realize combinational alternating logic, they proposed using self-dual functions (a function for which the normal output is the complement of the output when the inputs are complemented). Yamamoto, Watanabe, and Urano [YAMA] independently developed alternating logic and did some original work on its single error detection properties.

The earliest research specifically on formal design of self-checking networks was done in 1968 by Carter and Schneider [CART]. In 1971, the model was improved and the theory of self-checking was developed by Anderson and Metze [ANDE1,ANDE2]. Reynolds and Metze [REYN1] extended the work in alternating logic and developed a formal description of self-checking alternating logic. Reynolds [REYN2,REYN3] also proposed an approach for sequential network design and established conditions for alternating logic primitives to form a complete gate set.

2. BASIC CONCEPTS OF SELF-CHECKING ALTERNATING LOGIC (SCAL)

2.1. Introduction

In order to understand the exposition of the results in SCAL discussed in later chapters, a knowledge of the basic concepts in SCAL is necessary. This chapter will present some important definitions and theorems about SCAL and will discuss the merits of SCAL. A discussion of many of these topics is also given by Reynolds [REYN1]. The abbreviations, conventions, and nomenclature used throughout the thesis are given in Appendix 1. The word "function" will be used to refer to the logical operation being performed. A network is an implementation of a function, and a system is a combination of networks.

2.2. Definitions

In designing a self-checking system, it is necessary to specify the types of physical failures against which the system is to be protected. The most common model of failures is one which assumes a single fault in a network's logic operation. This model assumes that only one failure in the system occurs and that the failure causes the logic value of one line to be stuck-at 0 (s/0) or stuck-at 1 (s/1). Field experience has shown this model to be a good one if the test for the failure is applied reasonably soon after the failure so that a second failure is unlikely. In addition, it is assumed that the network is free of faults when it is initially used.

For most of this thesis, the single fault model will be used. In analyzing a network logic diagram it will be helpful to use a formal definition.

Definition 2.1. A single fault is a network condition in which one line is s/0 or s/1.

The line may be stuck either permanently or temporarily; i.e., transient failures are included. The transient failure may or may not be observable. If the fault does not affect the output for the input(s) during which the transient failure occurs, then no fault would be observed. If the single fault is permanent, and if the line is not redundant, then some input to the network exists such that the fault is observable. A redundant line is defined here as a line which may be removed from the network without affecting its operation. The complications of multiple line redundancies are deferred to previous work by Smith and Metze [SMIT1].

Other fault models include the unidirectional fault model and the multiple fault model. The unidirectional fault model is used extensively in other work on self-checking networks [ANDE1,SMIT2]. The multiple fault model is often used in static network diagnosis [CHA].

Definition 2.2. A unidirectional fault is a condition in which any number of lines is stuck at one value.

Definition 2.3. A multiple fault is a condition in which more than one line is stuck.

A single fault is a degenerate example of a unidirectional fault and a unidirectional fault is a degenerate example of a multiple fault. Examples of the representation of these faults are given in Figure 2.1.

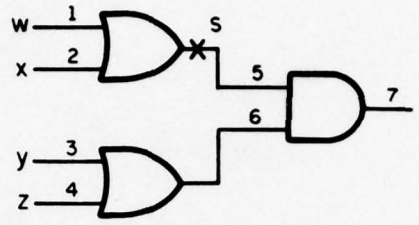


Figure 2.1a. Single fault: line 5 stuck-at value s, $s \in (0,1)$

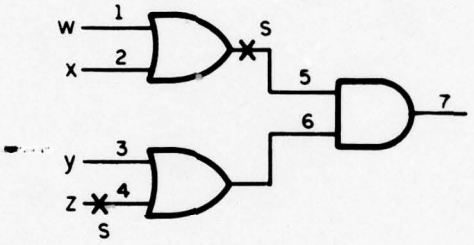
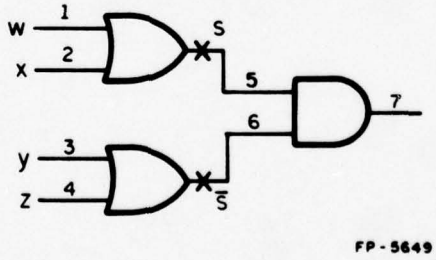


Figure 2.1b. Unidirectional fault: lines 4 and 5 stuck-at value s, $s \in (0,1)$



FP-5649

Figure 2.1c. Multiple fault: line 5 stuck-at value s and line 6 stuck-at value $\bar{s}$, $\bar{s} \in (0,1)$

A network which is dynamically checked for failures is referred to as self-checking. For this thesis, self-checking will be defined using the single fault model. Although the system is also self-checking for many multiple faults, the fault coverage is complete only for single faults. In the definition of self-checking, let f represent the single fault, X the network input vector, F(X) the normal network output, and $F_f(X)$ the network output when fault f occurs. It is assumed F(X) is a single output network, although the analysis is easily extended to multiple outputs.

Definition 2.4. A self-checking network is a network which satisfies the two constraints (with input X different from input Y):

(a) $\forall f, \exists X \ni F(X) \neq F_f(X)$

(b) $\nexists f \ni \{\exists [X \in \{\text{code inputs}\} \ \& \ Y \in \{\text{code inputs}\}] \ni [F_f(X) = F(Y)]\}$

Condition (a) will be referred to as the self-testing requirement and condition (b) will be referred to as the fault-secure requirement. The definition of self-testing normally requires that $F_f(X) \notin \{\text{all } F(X)\}$. However, Smith [SMIT2] has shown that this standard definition overlaps with the fault-secure definition. Therefore, the revised definition of self-testing used in part (a) of Definition 2.4 will be used.

Alternating logic is the most well known and probably the simplest form of time redundancy used in hardware dynamic fault detection. An alternating network is used to implement alternating logic.

Definition 2.5. An alternating network is a network which for an input sequence $(X,\overline{X})$ generates an output sequence $(F(X),F(\overline{X}))$ such that $F(\overline{X})=\overline{F}(X)$.

It may be useful in digital controllers to have an alternating signal output, but primary use of the alternating logic is probably in building self-checking systems. The network used to implement these types of systems is called a SCAL network. Self-checking alternating logic may now be formally defined.

Definition 2.6. A SCAL network is an alternating network which is self-checking.

In order to satisfy the requirements for a network to implement alternating logic, certain requirements are placed on the function being realized. First it is necessary to define a self-dual function.

Definition 2.7. A self-dual function is a function such that $F(\overline{X})=\overline{F}(X)$.

Using the definitions given, it is now possible to present some of the basic theorems in SCAL.

2.3. Theorems on SCAL

Reynolds [REYN1] presented the following theorem to describe the requirements for an alternating network to realize a function.

Theorem 2.1. *A network realizing a function F is an alternating network iff F is a self-dual function.*

Proof: If F is self-dual then $F(\overline{X})=\overline{F}(X)$ for every X and any network realizing F must therefore be an alternating network. Conversely, if a network is an alternating network, then $F(\overline{X})=\overline{F}(X)$ and F is self-dual. Q.E.D.

It has been shown [YAMA] that any network can be made self-dual with the addition of only one input. The additional input is the period clock which is 0 in the first period when the true input is applied and 1 in the second period when the complemented input is applied. The period clock will be represented by ϕ . The requirements for an alternating network to be a SCAL network may be determined using the previous definitions and theorem.

Theorem 2.2. An alternating network is a SCAL network iff the function realized satisfies two conditions (with input sequence $(Y,\overline{Y})$ different from input sequence $(X,\overline{X})$):

(a) $\forall f \exists X \ni (F(X),F(\overline{X})) \neq (F_f(X),F_f(\overline{X}))$

(b) $\nexists f \ni \{\exists[(X,\overline{X}) \;\&\; (Y,\overline{Y})] \ni [(F_f(X),F_f(\overline{X})) = (F(Y),F(\overline{Y}))]\}$

Proof: The conditions of Theorem 2.2 are both necessary and sufficient for alternating logic to satisfy the constraints of Definition 2.4. $(F(X),F(\overline{X}))$ is the normal output sequence for input sequence $(X,\overline{X})$ and $(F_f(X),F_f(\overline{X}))$ is the output sequence under fault f. Therefore condition (a) of the theorem is the alternating logic representation of part (a) of Definition 2.4. Also, $(X,\overline{X})$ and $(Y,\overline{Y})$ are alternating logic code inputs, so condition (b) of the theorem is the alternating logic representation of part (b) of Definition 2.4.

Q.E.D.

2.4. Merits of SCAL

SCAL has a limited range of applications for which it is desirable. Its primary advantages are: (1) some basic functions are already self-dual and involve no hardware cost to implement as SCAL (for example, see the optimal adder [LIU] in Figure 2.2), (2) in many other cases it requires less hardware

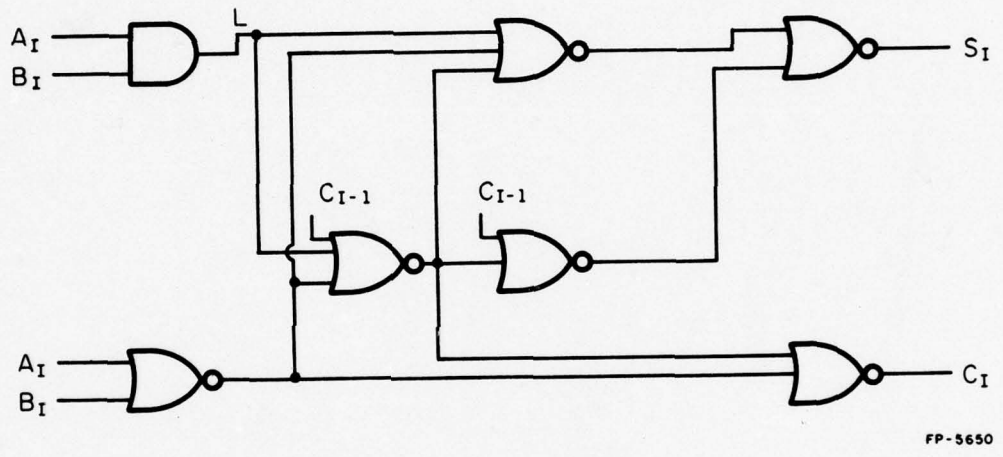


Figure 2.2. Self-dual adder

than other designs currently available for self-checking systems [REYN1], (3) it provides self-checking for single faults, and (4) since the redundancy is in time instead of space, no additional connections are required for the alternating logic modules. The primary disadvantages of SCAL are: (1) it requires twice as much time to perform the operation--this is not significant in systems with spare time, (2) it requires more hardware than networks which are not self-checking, and (3) not all failures are covered. In summary, if it is desirable to have a self-checking system and time is available at low system cost, then alternating logic is a good method of design.

3. COMBINATIONAL SCAL

3.1. Introduction

To design SCAL networks it is helpful to have design rules. Theorem 2.1 states that for a network to be an alternating network it must implement a *self-dual function*. *Theorem 2.2 gives the requirements for the alternating* network to be a SCAL network. To determine whether a combinational network satisfies the requirements of Theorem 2.2 some analysis procedures will be presented in this chapter. In the first three sections the networks will be assumed to have a single output. Section 3.4 will consider multiple output networks. Section 3.5 will present an algorithm for analyzing combinational networks to determine whether they are SCAL networks. An example of the application of the algorithm will be given in Section 3.6.

3.2. General Self-Checking Requirements

To simplify the presentation of results, the following symbolism will be used:

X represents an arbitrary input vector.

G(X) represents the value of an arbitrary line g for input X

F(X,G(X)) represents the network output for input X, with line g having value G(X).

F(X,s) represents the network output for input X, with line g stuck-at s, $s \in (0,1)$.

For correct operation, the alternating output is represented by:

$$(F(X,G(X)),F(\bar{X},G(\bar{X}))) = (F(X,G(X)),\bar{F}(X,G(X)))$$

In order to determine whether a network is self-checking, by Definition 2.4 it is necessary for the network to be self-testing and fault secure. To provide a procedure to determine whether a network satisfies these conditions, it will be helpful to know whether a given line causes the network to violate one of the conditions.

Theorem 3.1. An alternating logic network will generate an incorrect alternating output iff there exists a line g such that:

$$[F(X,G(X)) \neq F(X,s)] \;\&\; [F(\overline{X},G(\overline{X})) \neq F(\overline{X},s)] = 1$$

Proof: When X is the input and line g is stuck-at s then the equation above means that the output is the opposite of what it should be in the first time period. Also, when $\overline{X}$ is applied, the output of the faulty network is the opposite of what it should be in the second time period. Let Y be the value of $F(X,G(X))$. Then $F(X,s) \neq F(X,G(X)) \Rightarrow F(X,s) = \overline{Y}$ and $F(\overline{X},G(\overline{X})) = \overline{F}(X,G(X)) = \overline{Y}$, so $F(\overline{X},s) \neq F(\overline{X},G(\overline{X})) \Rightarrow F(\overline{X},s) = Y$. Therefore, the output is $(\overline{Y},Y)$ an incorrect alternating output, since the output should be $(Y,\overline{Y})$.

Conversely, if $F(X,G(X)) \neq F(X,s)$, but $F(\overline{X},G(\overline{X})) = F(\overline{X},s)$, then $F(\overline{X},G(\overline{X})) = \overline{Y} \Rightarrow F(\overline{X},s) = \overline{Y}$ and the output would be $(\overline{Y},\overline{Y})$ which is a nonalternating output. Similarly when $F(X,G(X)) \neq F(\overline{X},s)$, but $F(X,G(X)) = F(X,s)$, then the output would be a nonalternating (Y,Y). Therefore only if the conditions given exist will an incorrect alternating output be generated. Q.E.D.

Using Theorem 3.1 tests can be derived to detect stuck-at faults on line g. The analysis proceeds as follows:

Let: $A = F(X,0) \oplus F(X,G(X))$

$B = F(\bar{X},0) \oplus F(\bar{X},G(\bar{X}))$

$C = F(X,1) \oplus F(X,G(X))$

$D = F(\bar{X},1) \oplus F(\bar{X},G(\bar{X}))$

$E = A \ \& \ B$

$F = C \ \& \ D$

Theorem 3.2. Using the above symbol definitions, iff E=0 can line g be tested for stuck-at 0 faults and iff F=0 can line g be tested for stuck-at 1 faults. If E=0 then [A V B] are tests for stuck-at 0 faults. If F=0 then [C V D] are tests for stuck-at 1 faults.

Proof: Let $E = 0$

then $A \ \& \ B = 0$

and $\overline{A \ \& \ B} = 1$

and $\bar{A} \vee \bar{B} = 1$

and $[F(X,0) \oplus F(X,G(X)) = 0] \vee [F(\bar{X},0) \oplus F(\bar{X},G(\bar{X})) = 0] = 1$

and $[F(X,0) = F(X,G(X))] \vee [F(\bar{X},0) = F(\bar{X},G(\bar{X}))] = 1$

Letting $F(X,G(X)) = Y \Rightarrow F(\bar{X},G(\bar{X})) = \bar{Y}$.

This implies $[F(X,0) = Y] \vee [F(\bar{X},0) = \bar{Y}] = 1$.

So the output when g is stuck-at 0 for input $(X,\bar{X})$ should be $(Y,\bar{Y})$ and when E=0 it is $(Y,\bar{Y})$ or (Y,Y) or $(\bar{Y},\bar{Y})$. If it is required that [A V B =1], then:

$A \vee B = 1$

and $[F(X,0) \oplus F(X,G(X)) = 1] \vee [F(\bar{X},0) \oplus F(\bar{X},G(\bar{X})) = 1] = 1$

and $[F(X,0) \neq F(X,G(X))] \vee [F(\bar{X},0) \neq F(\bar{X},G(\bar{X}))] = 1.$

Again letting $[F(X,G(X)) = Y] \Rightarrow [F(\bar{X},G(\bar{X})) = \bar{Y}]$.

This implies $[F(X,0) = \bar{Y}] \vee [F(\bar{X},0) = Y] = 1$.

So the output for g stuck-at 0 for input $(X,\bar{X})$ when $[A \vee B = 1]$ is $(\bar{Y},Y)$ or $(\bar{Y},\bar{Y})$ or (Y,Y). Combining the results, when E=0 and inputs are applied so that $[A \vee B = 1]$, then the output for g stuck-at 0 is $(\bar{Y},\bar{Y})$ or (Y,Y). These are non-alternating outputs. This is detected and thus the fault is detected.

Conversely, assume by way of contradiction that E=1. Then the proof is:

$E = 1$

then $A \& B = 1$

and $[F(X,0) \neq F(X,G(X))] \& [F(\bar{X},0) \neq F(\bar{X},G(\bar{X}))] = 1$

Letting $F(X,G(X)) = Y \Rightarrow F(\bar{X},G(\bar{X})) = \bar{Y}$ and $[F(X,0) \neq Y] \& [F(\bar{X},0) \neq Y] = 1$.

This implies $[F(X,0) = \bar{Y}] \& [F(\bar{X},0) = Y] = 1$.

So for input $(X,\bar{X})$ the output should be $(Y,\bar{Y})$ but when E=1, the output is $(\bar{Y},Y)$, an incorrect alternating output. Since the output alternates, the fault cannot be detected. Therefore only if E=0 can the fault be detected.

The proof for the case of F=0 is similar. Q.E.D.

This is readily observed in the following example illustrated in Karnaugh map form in Figure 3.1, where

$F(X,G(X)) = G(X) \& x_3 \vee \bar{x}_1\bar{x}_2\bar{x}_3 \vee x_2\bar{x}_3x_4 \vee x_1\bar{x}_3\bar{x}_4$

and $G(X) = \bar{x}_1x_2\bar{x}_3 \vee \bar{x}_1x_2\bar{x}_4 \vee x_1\bar{x}_2x_3x_4$:

An algebraic analysis of A,B, and E gives:

$$A = F(X,0) \oplus F(X,G(X))$$

$$= (0 \vee \bar{x}_1\bar{x}_2\bar{x}_3 \vee x_2\bar{x}_3x_4 \vee x_1\bar{x}_3\bar{x}_4) \oplus$$

$$(G(X) \& x_3 \vee \bar{x}_1\bar{x}_2\bar{x}_3 \vee x_2\bar{x}_3x_4 \vee x_1\bar{x}_3\bar{x}_4$$

$$= G(X) \& x_3 = x_1\bar{x}_2x_3x_4 \vee \bar{x}_1x_2x_3\bar{x}_4$$

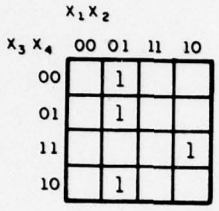


Figure 3.1a. G(X)

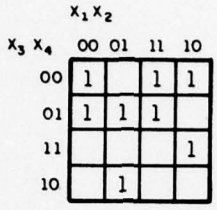


Figure 3.1b. F(X,G(X))

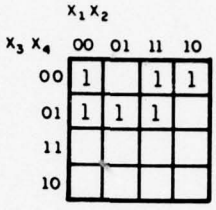


Figure 3.1c. F(X,0)

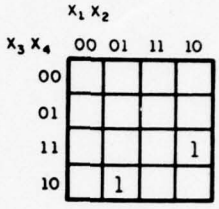


Figure 3.1d. A

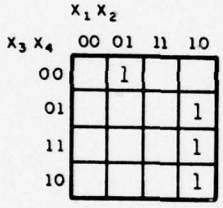


Figure 3.1e. G($\overline{X}$)

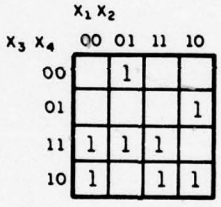


Figure 3.1f. F($\overline{X}$,G($\overline{X}$))

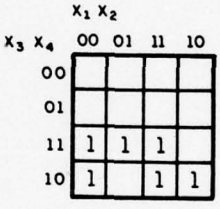


Figure 3.1g. F($\overline{X}$,0)

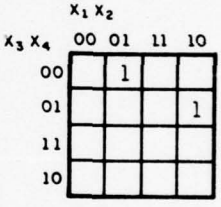


Figure 3.1h. B

FP-5683

$$B = F(\bar{X},0) \oplus F(\bar{X},G(\bar{X}))$$

$$= (0 \vee x_1x_2x_3 \vee \bar{x}_2x_3\bar{x}_4 \vee \bar{x}_1x_3x_4) \oplus$$

$$(G(\bar{X}) \,\&\, \bar{x}_3 \vee x_1x_2x_3 \vee \bar{x}_2x_3\bar{x}_4 \vee \bar{x}_1x_3x_4)$$

$$= G(\bar{X}) \,\&\, \bar{x}_3 = \bar{x}_1x_2\bar{x}_3\bar{x}_4 \vee x_1\bar{x}_2\bar{x}_3x_4$$

$$A \vee B = x_3(x_1\bar{x}_2x_4 \vee \bar{x}_1x_2\bar{x}_4) \vee \bar{x}_3(\bar{x}_1x_2\bar{x}_4 \vee x_1\bar{x}_2x_4)$$

$$E = A \,\&\, B = 0$$

By Theorem 3.2, since E=0, then inputs for [A V E] are test for line g stuck-at 0. The tests are: 1011, 0110, 0100, 1001. It is not necessary to know the correct output values for these inputs since their complement is also applied as input. When the input and its complement is applied, the output will alternate if there is no fault and will not alternate if line g is stuck-at 0. As for all self-checking networks, the checker determines if the output is correct. For testing purposes, whichever input of the input pair is applied first is irrelevant. The pairs are: (1011,0100) and (0110,1001).

Although analysis of this particular case is lengthy, the algorithm provides a structured approach for computer computation. Similarly, the case of g stuck-at 1 can be analyzed.

A method has been given to determine whether particular inputs check whether the network is self-checking for faults on specific lines. It may turn out that there does not exist an input which checks for certain faults. When this happens the network is determined to not be self-checking.

Theorem 3.3. Using the previous symbol definitions, if for any line g, $[\partial X_i \partial E = 0]$ or $[\partial X_i \partial F = 0]$, then the network is not self-checking.

Proof: From Definition 2.4, to be self-checking it is required that there be a test for every potential fault. Each line may fail and so become stuck-at 0 or stuck-at 1. Theorem 3.2 specifies that the condition E=0 must be satisfied for line g stuck-at 0 to be tested, and the condition F=0 must be satisfied for line g stuck-at 1 to be tested. If there is no input X_i such that the condition required for a fault on line g to be tested is satisfied, then the network is not self-checking with respect to that line. Therefore the network is not self-checking. Q.E.D.

Furthermore, if a line in the network cannot be tested for either a stuck-at 0 or a stuck-at 1 fault, then the network is independent of the logic value of the line.

Theorem 3.4. Using the previous symbol definitions, for a given line g, if [A V C = 0] , then the line is redundant.

Proof: Let [A V C = 0]. This implies A=0 and C=0. A=0 implies $F(X_i,0)=F(X_i,G(X_i))$ for all X_i and C=0 implies $F(X_i,1)=F(X_i,G(X_i))$ for all X_i. So, [A V C =0] requires that the network output have the same value for either value of the line g for all inputs. Therefore the value of the line has no effect on the output and so is redundant. Q.E.D.

In addition, if a line can be tested for a fault in only one direction, say s, then the subnetwork generating the line value may be removed and replaced by a constant input of $\bar{s}$. In further analysis it will be assumed that all such replacements have been done. The fault model for the line includes only stuck-at s faults, since stuck-at $\bar{s}$ is the correct operation of the line.

It is necessary to consider the problem of a multiple line redundancy, where a combination of several lines together are redundant, but any one is not redundant by itself. Smith and Metze [SMIT1] discuss this problem, but for the discussion here it is assumed that all the redundancies are single line redundancies. The redundancies will also be asssumed to be unintentional, i.e., not intended for such purposes as protecting from sequential logic hazard conditions.

Theorem 3.5. If a self-dual network is irredundant, then it is self-testing.

Proof: If the network is irredundant, then there does not exist any line in the network which satisfies the equation [A V C = 0] of Theorem 3.4. Since E = A&B , then A=0 implies E=0. Similarly C=0 implies F=0. So by Theorem 3.2 any line can be tested. Assuming all inputs are applied at some time, the network is self-testing. Q.E.D.

In further analysis it will be assumed the networks are irredundant, all inputs are applied, and hence the networks are self-testing. Therefore to satisfy the conditions of Definition 2.4 for a network to be self-checking, it will only be necessary to consider whether it is fault-secure.

To determine whether a network line prevents the network from being self-checking, the equation in Theorem 3.1 is evaluated. To simplify the analysis, a corollary to Theorem 3.1 can be derived.

The equation in Theorem 3.1 is:

$$[F(X,G(X)) \neq F(X,s)] \;\&\; [F(\bar{X},G(\bar{X})) \neq F(\bar{X},s)] = 1$$

By algebra, this translates to:

$$[F(X,G(X)) \;\&\; \bar{F}(X,s) \vee \bar{F}(X,G(X)) \;\&\; F(X,s)]$$
$$\&\; [F(\bar{X},G(\bar{X})) \;\&\; \bar{F}(\bar{X},s) \vee \bar{F}(\bar{X},G(\bar{X})) \;\&\; F(\bar{X},s)] = 1 \;.$$

Since alternating logic is used, by Definition 2.5,

$$F(\overline{X}) = \overline{F}(X) \quad \text{or} \quad F(\overline{X},G(\overline{X})) = \overline{F}(X,G(X))$$

Using this relation, the above equation changes to:

$$[F(X,G(X)) \,\&\, \overline{F}(X,s) \vee \overline{F}(X,G(X) \,\&\, F(X,s)] \,\&\, [F(X,G(\overline{X})) \,\&\, \overline{F}(\overline{X},s) \vee F(X,G(X)) \,\&\, F(\overline{X},s)] = 1$$

By algebra, this reduces to:

$$F(X,G(X)) \,\&\, \overline{F}(X,s) \,\&\, F(\overline{X},s) \vee \overline{F}(X,G(X)) \,\&\, F(X,s) \,\&\, \overline{F}(\overline{X},s) = 1 \ .$$

Since alternating logic is used, whenever X_i is applied, $\overline{X}_i$ is also applied in the alternating pair $(X_i,\overline{X}_i)$. So if for some alternating pair $(X_i,\overline{X}_i)$ for input X_i,

$$\overline{F}(X_i,G(X_i)) = 1 \quad \text{and} \quad F(X_i,s) = 1 \quad \text{and} \quad \overline{F}(\overline{X}_i,s) = 1$$

then for another alternating pair $(X_j,\overline{X}_j)$ with $X_j=\overline{X}_i$ and $\overline{X}_j=X_i$

$$\overline{F}(\overline{X}_j,G(\overline{X}_j)) = F(X_j,G(X_j)) = 1 \quad \text{and} \quad F(\overline{X}_j,s) = 1 \quad \text{and} \quad \overline{F}(X_j,s) = 1$$

So if all input pairs are applied, then if for some X

$$\overline{F}(X,G(X)) \,\&\, F(X,s) \,\&\, \overline{F}(\overline{X},s) = 1$$

then also for another X

$$F(X,G(X)) \,\&\, \overline{F}(X,s) \,\&\, F(\overline{X},s) = 1 \ .$$

So only one of the above products needs to be checked to determine the self-checking characteristics. To analyze for both stuck-at 0 and stuck-at 1 faults, s is replaced by the appropriate value as the following corollary of Theorem 3.1 states.

Corollary 3.1. The network is self-checking with respect to a line g iff line g is in an irredundant self-dual network and satisfies

$$F(X,G(X)) \,\&\, [\overline{F}(X,0) \,\&\, F(\overline{X},0) \vee \overline{F}(X,1) \,\&\, F(\overline{X},1)] = 0 \ .$$

Corollary 3.1 may be broken down to consider separately stuck-at 0 and stuck-at 1 faults in line g. The network is self-checking with respect to stuck-at 0 faults in line g iff

$$F(X,G(X)) \;\&\; \overline{F}(X,0) \;\&\; F(\overline{X},0) = 0 \ .$$

The network is self-checking with respect to stuck-at 1 faults in line g iff

$$F(X,G(X)) \;\&\; \overline{F}(X,1) \;\&\; F(\overline{X},1) = 0 \ .$$

The equation in the corollary may also be written in different forms taking advantage of the equivalences stated before. These revised equations are:

$$F(X,G(X)) \;\&\; \overline{F}(X,0) \;\&\; F(\overline{X},0) \vee F(\overline{X},G(\overline{X})) \;\&\; F(X,1) \;\&\; \overline{F}(\overline{X},1) = 0$$

$$F(\overline{X},G(\overline{X})) \;\&\; F(X,0) \;\&\; \overline{F}(\overline{X},0) \vee F(X,G(X)) \;\&\; \overline{F}(X,1) \;\&\; F(\overline{X},1) = 0$$

$$F(\overline{X},G(\overline{X})) \;\&\; [F(X,0) \;\&\; \overline{F}(\overline{X},0) \vee F(X,1) \;\&\; \overline{F}(\overline{X},1)] = 0$$

In some cases it may be simpler to evaluate these equations rather than the one given in Corollary 3.1.

3.3. Sufficient Self-Checking Conditions

Rather than having to examine every line in a network to determine whether the conditions of Corollary 3.1 hold, it is possible to determine and use some general rules for deciding whether certain types of lines always satisfy the conditions of Corollary 3.1.

Theorem 3.6. If $\forall X, G(X) \neq G(\overline{X})$, then the network is self-checking with respect to line g.

Proof: Consider an input X_i with $G(X_i) = d$, $d \in (0,1)$. Then by the assumption in Theorem 3.6, $G(\overline{X}_i)=\overline{d}$. So $F(\overline{X}_i,G(X_i))=F(X_i,d)$ and $F(\overline{X}_i,G(\overline{X}_i))=F(\overline{X}_i,\overline{d})$. Now

let $F(X_i,d)=Y$ and $F(X_i,\overline{d})=\overline{Y}$. If g is stuck-at s, then $G(X_i)=s$ and $G(\overline{X}_i)=s$. By Theorem 3.1, the network will not be self-checking iff

$$[F(X,G(X)) \neq F(X,s)] \;\&\; [F(\overline{X},G(\overline{X})) \neq F(\overline{X},s)] = 1$$

By substitution of the values above, the equation changes to $[Y \neq F(X,s)] \;\&\; [\overline{Y} \neq F(\overline{X},s)] = 1$. . Either s=d or $s=\overline{d}$; without loss of generality, let s=d. So, $F(X,s)=F(X,d)=Y$ and the equation of Theorem 3.1 is not satisfied. Therefore, an irredundant self-dual network is self-checking with respect to line g. Q.E.D.

By this theorem, a network is self-checking with respect to all lines which alternate in the two time periods of alternating logic. Using Theorem 3.6, the result presented by Reynolds [REYN1] that a network is self-checking with respect to all input lines can be easily proven since all input lines have alternating values.

If a line does not fan out, then the analysis to determine if the network is self-checking with respect to the line is simpler than if it did fan out.

Theorem 3.7. If a line g in a self-dual network does not fan out in its path to the output and the gates in the path are all unate, then the network is self-checking with respect to line g.

Proof: For some input X_i the line g must be sensitized if the network is irredundant. This will force the output to be the opposite of what it would otherwise be, $F(X_i,s)=\overline{F}(X_i,G(X_i))$. Since the network elements on the path from line g to the output are all monotonic and there is only one such path to the output, g can only affect the output in one direction when it is stuck-at s. Therefore, line g is not sensitized when $\overline{X}_i$ is applied. If line g is not

sensitized, then the output would be unchanged and a nonalternating output pair $(\overline{F}(X_i,G(X_i)),F(\overline{X}_i,G(\overline{X}_i))) = (\overline{F}(X_i,G(X_i)),\overline{F}(X_i,G(X_i)))$ is obtained. Therefore, the failure is detected and the network is self-checking with respect to line g. Q.E.D.

If network elements that were not monotonic were allowed on the path from line g to the output, then Theorem 3.7 would not apply. For example, consider Figure 3.2 where an exclusive-or gate is on the path from line g to the output. If the output of g for $(X_i,\overline{X}_i)$ is (0,0), then when g is stuck-at 1 the output of g changes to (1,1). If the output of Q for input $(X_i,\overline{X}_i)$ is (0,1), then normally the output of f is (0,1). However when g is stuck-at 1, then the output is (1,0) and the failure of line g is undetected. In this case the network is not self-checking with respect to line g stuck-at 1.

From Theorem 3.7 the result of Yamamoto, Watanabe, and Urano [YAMA] that two-level self-dual networks with monotonic gates are self-checking can be easily proven. Since none of the non-input lines fan out in the single output network, the conditions of Theorem 3.7 are satisfied by the non-input lines. The input lines alternate and so the network is self-checking with respect to all the lines in a two level network. A level of inverters on the inputs preserves this property since their outputs alternate. Furthermore, the analysis can be applied to each output of a multiple output network. If each output is checked for alternation, then the multiple output network is self-checking.

The following result is based on the work of Reynolds [REYN1].

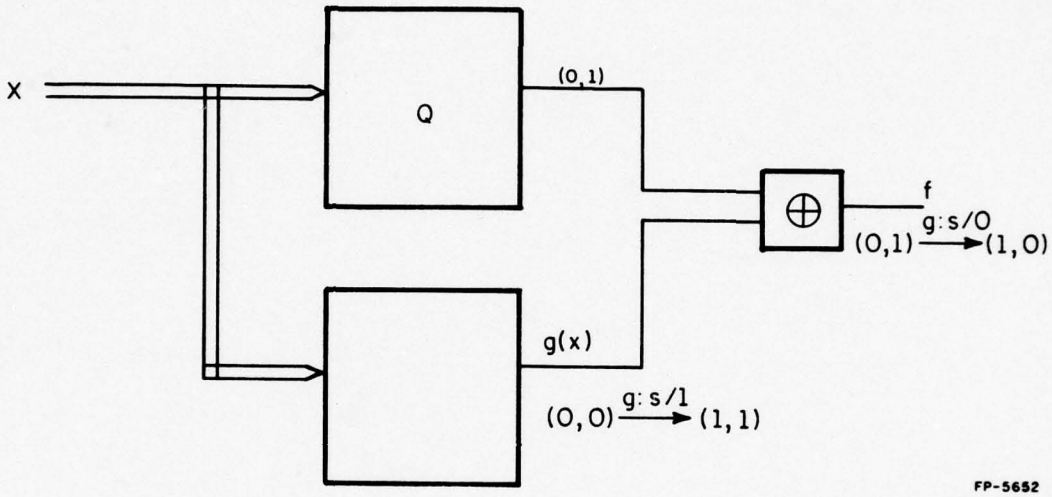


Figure 3.2. Incorrect alternation

Definition 3.1. Path parity from line g to the network output is the modulo 2 number of inversions on the path.

Theorem 3.8. If all paths from line g to the network output have the same parity, then the network is self-checking with respect to line g.

In multiple level networks the input is often fed to a level other than the first. This enhances the self-checking properties of an irredundant network as the following theorem proves. This theorem applies to a restricted set of gate types defined as follows:

Definition 3.2. A standard gate is a NOT,NAND,AND,NOR, or OR gate.

Theorem 3.9. In an irredundant network, if a line g is the input to the same standard gate as an alternating line, then the network is self-checking with respect to line g.

Proof: Since the network is irredundant and all inputs are presumed to be applied, then the network is self-testing with respect to all the lines, including line g, by Theorem 3.5. Now consider the fault secure property. In a standard gate one input value dominates by forcing the output to a specific value when the dominant input value is applied. This is observed by considering their truth tables. The dominant input values are 0 for NAND and AND, and 1 for NOR and OR. Since an alternating value is applied to one input line of the standard gate, then for the time period the dominant alternating input is applied, the output value is independent of the other inputs to the gate. Even if one of the other inputs is stuck-at s, the output of the gate will be correct. Therefore, the network output will be correct for that time

period. During the other time period, if the network output is correct, then the fault does not affect the output. If the network output is incorrect, then it is nonalternating and the fault is detected. Either way, the network is fault secure. Since the network is both fault secure and self-testing, by Definition 2.4 it is self-checking. Q.E.D.

Note that Theorem 3.9 would not apply to a gate which is not a standard gate, such as the XOR gate or majority gate since these do not exhibit the dominance property.

Using the results presented it is now possible to follow a procedure to determine whether a self-dual combinational single-output network is a SCAL network. If a line passes any of the following tests, then an irredundant self-dual network is self-checking with respect to that line:

1. It alternates in value for all X_i.
2. It does not fan out and its path to the output is through unate gates.
3. Path parity is the same for all paths from the line to the network output.
4. It is the input to the same standard gate as an alternating line.
5. Conditions of Corollary 3.1 are met.

3.4. Multiple Output Combinational SCAL

So far only single output networks have been considered. In this section a procedure is presented for determining whether a multiple output network is self-checking with respect to a line in the network.

In general, multiple output networks may be classified as being of two types: separable (Figure 3.3a) and nonseparable (Figure 3.3b). As is implied by the names, the two types differ according to whether the combinational logic generating the outputs from the alternating inputs may be entirely separated or not. If the combinational logic may be separated, then the analysis of preceding sections may be applied to the logic generating each output individually to determine whether the network is self-checking. Similarly when the outputs are not independent but a portion of their logic is, then the methods of preceding sections may be applied to the portion of the logic that is independent and thus its effect on the self-checking property of the network may be assessed.

However, when some logic is shared to generate a line common to more than one output (as line g in Figure 3.3b is), then the analysis procedure must be changed.

Definition 3.3. A multiple output network is self-checking with respect to a line shared commonly between more than one output iff for any input X_i for which an output alternates incorrectly, some other output does not alternate for that input.

To evaluate whether a line commonly shared between more than one output satisfies the self-checking requirements, a determination of the specific requirements on the line is useful. The symbol definitions used in the previous sections will be used with a subscript added to indicate a specific output line.

Theorem 3.10. For a line g stuck-at s, if an input X_i causes

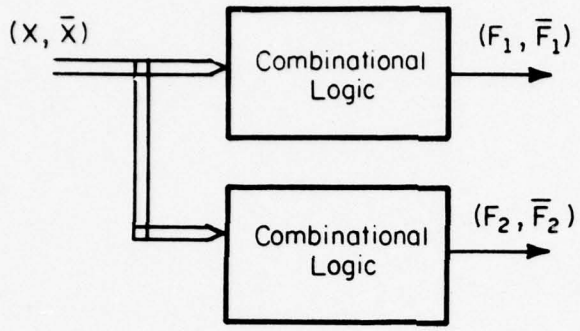


Figure 3.3a. Separable multiple output network

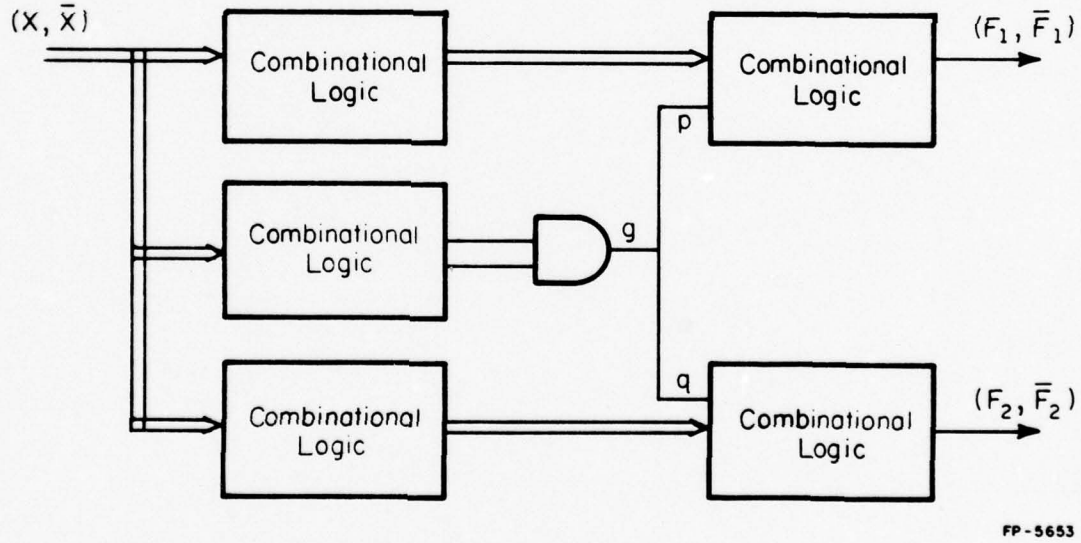


Figure 3.3b. Nonseparable multiple output network

$F_j(X_i,s)=\overline{F}_j(X_i,G(X_i))$ and $F_j(\overline{X}_i,s)=\overline{F}_j(\overline{X}_i,G(\overline{X}_i))$ then some output F_k must satisfy $F_k(X_i,s)=F_k(\overline{X}_i,s)$ for the network to be self-checking.

Proof: By Definition 3.3 an incorrect alternating output on one network output, F_j, must be accompanied by a nonalternating output on another network output, F_k, if the network is to be self-checking with respect to a fault on line g. Q.E.D.

If the network is shown by Theorem 3.10 to be self-checking with respect to line g, it may still not be self-checking with respect to lines p or q of Figure 3.3b. These two lines must be checked as all other lines are checked which are used to generate only one network output.

From Theorem 3.10, it is observed that the requirements for lines which feed more than one network output are less strict than for other network lines which feed only one network output. In the example in Section 3.6 this will be observed. Also, for line L of the adder in Figure 2.2, it was necessary that only the reduced requirements for multiple output networks be satisfied. Otherwise, the adder would not be self-checking.

In the analysis, all lines in the combinational logic block generating g in Figure 3.3 are checked under the reduced requirements for multiple outputs. To evaluate whether the network is self-checking with respect to a line g shared by more than one output, first a single output is checked to determine whether it is self-checking with respect to line g. If all outputs which use line g are self-checking with respect to line g, then the network is self-checking with respect to line g by Definition 3.3. However, if one output is not self-checking with respect to line g, the network may still be self-checking with respect to line g according to Theorem 3.10. To provide a

specific analysis procedure to check if this is so, Corollary 3.2 is derived.

Consider a stuck-at fault, s, on line g, which causes an incorrect alternating output on output F_1. The equation of Corollary 3.1 is then

$$F_1(X,G(X)) \;\&\; \bar{F}_1(X,s) \;\&\; F_1(X,s) \neq 0$$

If another output, F_k, has a nonalternating output for this fault then $F_k(X,s) = F_k(\bar{X},s)$ or $[\bar{F}_k(X,s) \;\&\; \bar{F}_k(\bar{X},s) \vee F_k(X,s) \;\&\; F_k(\bar{X},s)] = 1$.

If the value(s) of X_i for which the incorrect alternating output is generated on F_1 are included among the outputs of F_k which are nonalternating, then by Theorem 3.10 the network is self-checking with respect to g. Furthermore, if there are several outputs which share line g, then all the inputs for which they are nonalternating are protected from not being self-checking on output 1. This is stated more precisely in Corollary 3.2:

Corollary 3.2. If $F_1(X,G(X)) \;\&\; \bar{F}_1(X,s) \;\&\; F_1(\bar{X},s) \neq 0$ for some output, but

$$\left[\sum_{k=1}^{n} \bar{F}_k(X,s) \;\&\; \bar{F}_k(\bar{X},s) \vee F_k(X,s) \;\&\; F_k(\bar{X},s\right] \;\&\; F_1(X,G(X)) \;\&\; \bar{F}_1(X,s) \;\&\; F_1(\bar{X},s) = 0$$

then the n-output irredundant self-dual network is self-checking with respect to line g by checking all the outputs.

In the analysis of networks to determine whether they are self-checking with respect to a line g shared commonly by more than one output, the single output analysis summarized in Section 3.3 will be done first since it is simplest. However if g does not satisfy at least one of these conditions, then it will be examined to determine whether it passes the more relaxed requirements for multiple output network lines.

3.5. Self-Checking Design and Analysis Algorithm

The results of previous sections now provide a complete algorithm for determining whether an irredundant self-dual single or multiple-output network is self-checking. The algorithm proceeeds as follows:

Algorithm 3.1.

1. Each network output will be regarded as independent of the others. Each line which is used to generate an output will be examined to determine whether it satisfies at least one of the following conditions:

A. It alternates for every input pair $(X_i, \overline{X}_i)$.

B. It does not fan out and its path to the output is through unate gates.

C. Path parity is the same for all paths from the line to the network output.

D. It is the input to the same standard gate (NAND, NOR, OR, AND, or NOT) as an alternating line.

E. It meets the condition

$$F(X,G(X)) \;\&\; (\overline{F}(X,0) \;\&\; F(\overline{X},0) \vee \overline{F}(X,1) \;\&\; F(\overline{X},1)) = 0 \ .$$

2. If any line from a subnetwork used by more than one output fails to meet one of these conditions for an output 1, then it is checked to see if

$$\left[\sum_{k=1}^{n} \overline{F}_k(X,s) \;\&\; \overline{F}_k(\overline{X},s) \vee F_k(X,s) \;\&\; F_k(\overline{X},s)\right] \;\&\; F_1(X,G(X)) \;\&\; \overline{F}_1(X,s\ ) \;\&\; F_1(\overline{X},s) = 0$$

3. If the line does not meet at least one of the above conditions, then the network is not self-checking.

A detailed example is presented in Section 3.6.

From the presentation of determining whether a network is self-checking, a few design recommendations surface: (1) minimize fan out, particularly of unequal parity to the output, (2) use two levels (plus an inverter level) to automatically achieve self-checking, and (3) share logic between as many outputs as possible, since this reduces the requirements on the lines for the network to be self-checking.

3.6. Example of Self-Checking Analysis

To illustrate the application of the analysis procedure summarized in Section 3.5, an example of its operation will be given. The example is based on the multiple output network of Figure 3.4. The example is contrived to best demonstrate the algorithm's operation. The self-dual functions implemented are:

$$F_1 = \bar{A}\,\bar{C} \vee B\bar{C} \vee \bar{A}B ,$$

$$F_2 = A \oplus B \oplus C ,$$

$$F_3 = \text{MAJORITY } (A,B,C) = AB \vee BC \vee AB .$$

The first step involves considering each output network separately and determining whether the network output is self-checking with respect to the lines used in generating it. First consider output F_1:

1. The lines used by F_1 are shown in Figure 3.5a. They are: 1,2,3,4,5,6,8,9,10,11,12,13,18,19,24,28,29,33,38.

2. Equivalent pairs of lines are: (3,24), (8,10), (9,13), and (19,28). So the reduced set of lines to analyze is: 1,2,3,4,5,6,8,9,11,12,18,19,29,33,38.

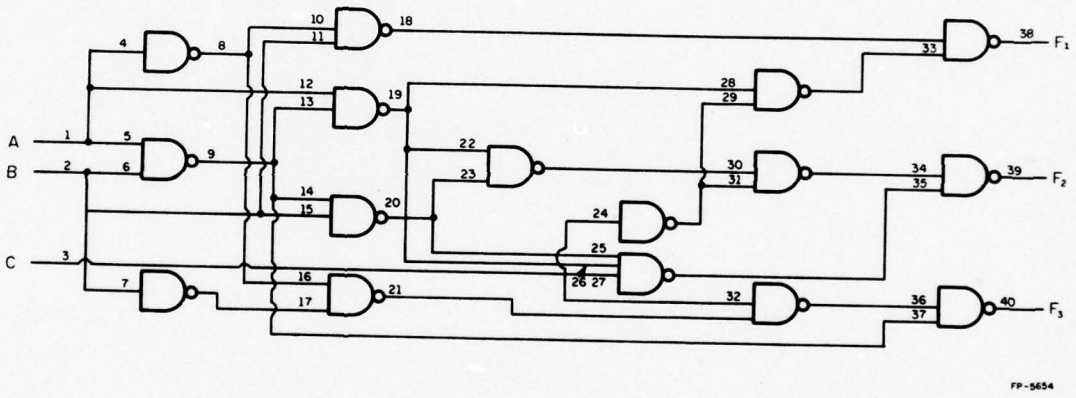


Figure 3.4. Multiple output network example

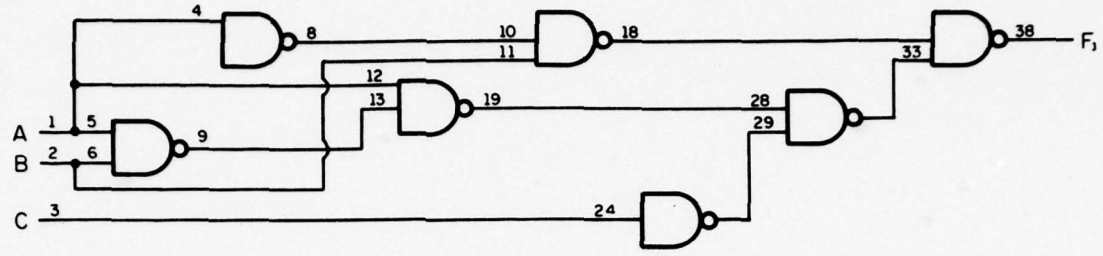


Figure 3.5a. Network for output F_1

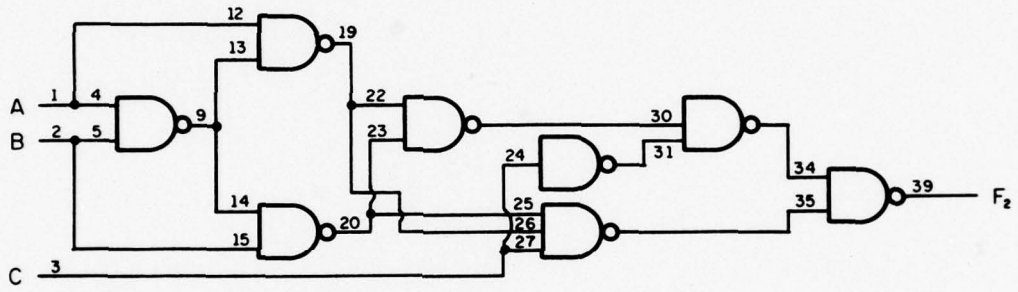


Figure 3.5b. Network for output F_2

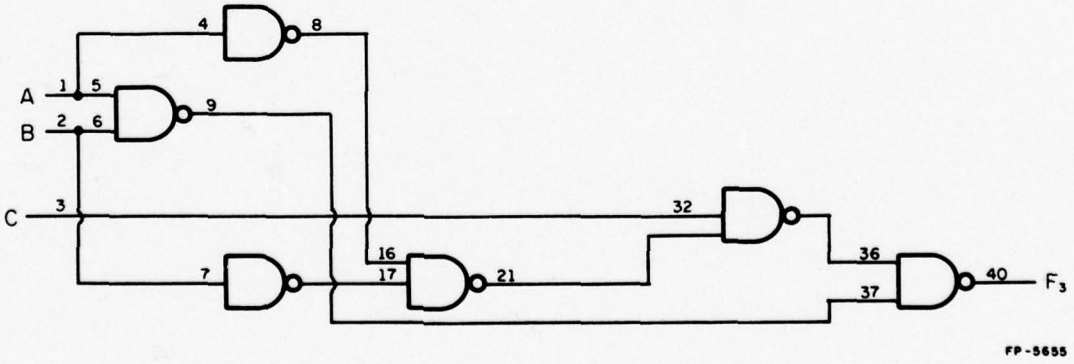


Figure 3.5c. Network for output F_3

3. Lines satisfying condition A of the algorithm are: 1,2,3,4,5,6,8,11,12,29,38. This leaves lines: 9,18,19,33.

4. Lines satisfying condition B of the algorithm are: 9,18,19,33. No lines remain which do not satisfy at least one of the conditions for the network to be self-checking. Therefore, the network is self-checking for output F_1.

Now consider output F_3:

1. The lines used by F_3 are shown in Figure 3.5c. They are: 1,2,3,4,5,6,7,8,9,16,17,21,32,36,37,40.

2. Equivalent lines are: (3,32), (8,16), and (9,37). So the reduced set of lines to analyze are: 1,2,3,4,5,6,7,8,9,17,21,36,40.

3. Lines satisfying condition A of the algorithm are: 1,2,3,4,5,6,7,8,17,40. This leaves lines: 9,21,36.

4. Lines satisfying condition B of the algorithm are: 9,21,36. There are no lines remaining which do not satisfy at least one of the conditions for the network to be self-checking. Therefore, the network is self-checking for output F_2.

Finally, consider the more complicated subnetwork which generates F_2:

1. The lines used by F_2 are shown in Figure 3.5b. They are: 1,2,3,4,5,9,12,13,14,15,19,20,22,23,24,25,26,27,30,31,34,35,39.

2. There are no equivalent lines, so all the lines need to be analyzed.

3. Lines satisfying condition A of the algorithm are: 1,2,3,4,5,12,15,24,27,31,39. The lines remaining to be analyzed are: 9,13,14,19,20,22,23,25,26,30,34,35.

4. Lines satisfying condition B of the algorithm are: 22,23,25,26,30,34,35. This leaves: 9,13,14,19,20.

5. No lines satisfy condition C of the algorithm.

6. Lines satisfying condition D of the algorithm are: 13,14. This leaves lines: 9,19,20.

7. Now evaluate each line to determine if condition E of the algorithm is satisfied: $F_2(X,9(X)) = 9 \;\&\; (A\,\overline{C} \vee B\,C) \vee \overline{9} \;\&\; C \vee \overline{A}\,\overline{B}\,\overline{C}$

$$F_2(X,0) = C;\; F(\overline{X},0) = \overline{C};\; \overline{F}(X,0) = \overline{C}$$

$$F_2(X,1) = A\overline{C} \vee B\,\overline{C} \vee \overline{A}\,\overline{B}\,C$$

$$F_2(\overline{X},1) = \overline{A}\,C \vee \overline{B}\,C \vee A\,B\,\overline{C}$$

$$\overline{F}_2(X,1) = A\,C \vee B\,C \vee \overline{A}\,\overline{B}\,\overline{C}$$

$$F_2(X,9(X)) \;\&\; F_2(\overline{X},0) \;\&\; \overline{F}_2(X,0) = \overline{A}\,B\,\overline{C} \vee A\,\overline{B}\,\overline{C}$$

$$F_2(X,9(X)) \;\&\; F_2(\overline{X},1) \;\&\; \overline{F}_2(X,1) = 0$$

So the condition E is not met by line 9 because the stuck-at 0 fault causes an incorrect alternating output when the input is $\overline{A}B\overline{C}$ or $A\overline{B}\overline{C}$.

$$F_2(X,19(X)) = 19 \;\&\; C \;\&\; (A \vee \overline{B}) \vee \overline{19} \;\&\; \overline{C} \vee \overline{A}\,B\,\overline{C}$$

$$F_2(X,0) = \overline{C};\; F_2(\overline{X},0) = C;\; \overline{F}_2(X,0) = C$$

$$F_2(X,1) = A\,C \vee \overline{B}\,C \vee \overline{A}\,B\,\overline{C}$$

$$F_2(X,1) = \overline{A}\,\overline{C} \vee B\,\overline{C} \vee A\,\overline{B}\,C$$

$$F_2(X,1) = A\,\overline{C} \vee \overline{B}\,\overline{C} \vee \overline{A}\,B\,C$$

$$F_2(X,19(X)) \;\&\; F_2(\overline{X},0) \;\&\; \overline{F}_2(X,0) = \overline{A}\,\overline{B}\,C \vee A\,B\,C$$

$$F_2(X,19(X)) \;\&\; F_2(\overline{X},1) \;\&\; \overline{F}_2(X,1) = 0$$

So the condition E is not met by line 19 because the stuck-at 0 fault causes an incorrect alternating output when the input is $\overline{A}\overline{B}C$ or ABC.

$$F_2(X,20(X)) = 20 \,\&\, C \,\&\, (\bar{A} \vee B) \vee \overline{20} \,\&\, C \vee A\bar{B}\bar{C}$$

$$F_2(X,0) = \bar{C};\ F(\bar{X},0) = C;\ \bar{F}(X,0) = C$$

$$F_2(X,1) = \bar{A}C \vee BC \vee A\bar{B}\bar{C}$$

$$F_2(\bar{X},1) = A\bar{C} \vee \bar{B}\bar{C} \vee \bar{A}BC$$

$$F_2(X,1) = \bar{A}\bar{C} \vee B\bar{C} \vee A\bar{B}C$$

$$F_2(X,20(X)) \,\&\, F_2(\bar{X},0) \,\&\, \bar{F}_2(X,0) = \bar{A}\bar{B}C \vee ABC$$

$$F_2(X,20(X)) \,\&\, F_2(\bar{X},0) \,\&\, \bar{F}_2(X,0) = 0$$

Again the condition E is not met by line 20 due to the stuck-at 0 fault causing an incorrect alternating output when the input is $\bar{A}\bar{B}C$ or ABC.

8. The lines 9,19,20 did not meet any of the conditions for a single output network to be self-checking; however, since 9 and 19 each go to more than one output, they may satisfy the more relaxed multiple output condition. They will each be separately analyzed to determine whether they satisfy the condition in Corollary 3.2.

First consider line 9 for the stuck-at 0 case. The output F_1 is also dependent on line 9:

$$F_1(X,9(X)) = \bar{A}B \vee \bar{A}\bar{C} \vee \bar{9}\,\bar{C}$$

The output F_3 is also dependent on line 9:

$$F_3(X,9(X)) = AC \vee BC \vee \bar{9}$$

So the summation term in Corollary 3.2 is:

$$\sum_{k=1}^{3} \bar{F}_k(X,s) \,\&\, \bar{F}_k(\bar{X},s) \vee F_k(X,s) \,\&\, F_k(\bar{X},s)$$

$$= (AC \vee \bar{B}C) \,\&\, (\bar{A}\bar{C} \vee B\bar{C}) \vee (\bar{C} \vee \bar{A}B) \,\&\, (C \vee A\bar{B}) \vee 0 \,\&\, 0 \vee 1 \,\&\, 1$$

$$= 0 \vee (A\bar{B}\bar{C} \vee \bar{A}BC) \vee 0 \vee 1 \qquad = 1$$

The whole equation in Corollary 3.2 is:

$$\bar{1} \;\&\; (\bar{A}\,B\,\bar{C} \vee A\,\bar{B}\,\bar{C}) = 0$$

Therefore the multiple output network is self-checking with respect to line 9. Now consider line 19 for the stuck-at 0 case. The only other output also dependent on 19 is output F_1.

$$F_1(X,19(X)) = 19 \;\&\; \bar{C} \vee \bar{A}\,B$$

So the summation term in Corollary 3.2 is:

$$\bar{F}_1(X,0) \;\&\; \bar{F}_1(\bar{X},0) \vee F_1(X,0) \;\&\; F_1(\bar{X},0) = (A \vee \bar{B}) \;\&\; (\bar{A} \vee B) \vee \bar{A}\,B \;\&\; A\,\bar{B}$$

$$= (\bar{A}\,\bar{B} \vee A\,B) \vee 0 = \bar{A}\,\bar{B} \vee A\,B$$

The whole equation in Corollary 3.2 becomes:

$$\overline{(\bar{A}\,\bar{B} \vee A\,B)} \;\&\; \bar{A}\,\bar{B}\,C \vee A\,B\,C) = (\bar{A}\,B \vee A\,\bar{B}) \;\&\; (\bar{A}\,\bar{B}\,C \vee A\,B\,C) = 0$$

Therefore the multiple output network is self-checking with respect to line 19.

Line 20 does not go to any other output, so the network is not self-checking with respect to line 20 stuck-at 0. The network has been shown to be self-checking for all other lines in this example. Some lines and their outputs for each input pair are shown in Figure 3.6. An X after the output pair means that the fault is detected by a nonalternating pair. Every output dependent on the line must exhibit this property for one input pair, since the network is irredundant and hence self-testing for alternating inputs. An * after the output pair means that an incorrect alternating pair is generated. For line 9, whenever this occurs on F_2 there is always a nonalternating output on F_1 or F_3. However for line 20 the incorrect alternating pair is not accompanied by a nonalternating pair on another output and so, as was shown earlier, the network is not self-checking with respect to line 20.

Line	Stuck-at /Normal	Output	Input Pairs: (000,111)	(001,110)	(010,101)	(011,100)
38	Normal	F_1	0,1	1,0	1,0	1,0
39	Normal	F_2	0,1	1,0	1,0	0,1
40	Normal	F_3	0,1	0,1	0,1	1,0
1	s/0	F_2	0,0X	1,1X	1,1X	0,0X
1	s/1	F_2	1,1X	0,0X	0,0X	1,1X
13	s/0	F_2	0,1	1,0	1,1X	0,0
13	s/1	F_2	0,0X	1,1X	1,0	0,1
22	s/0	F_2	1,1X	1,1X	1,0	0,1
22	s/1	F_2	0,1	1,0	1,0	0,0X
9	s/0	F_1	0,1	1,0	1,1X	1,0
9	s/1	F_1	0,0X	1,0	1,0	1,0
9	s/0	F_2	0,1	1,0	0,1*	1,0*
9	s/1	F_2	0,0	1,1X	1,0	0,1
9	s/0	F_3	1,1X	1,1X	1,1X	1,1X
9	s/1	F_3	0,1	0,0X	0,1	1,0
20	s/0	F_1	0,1	1,0	1,0	1,0
20	s/1	F_1	0,1	1,0	1,0	1,0
20	s/0	F_2	1,0*	0,1*	1,0	0,1
20	s/1	F_2	0,1	1,0	0,0X	1,1X
20	s/0	F_3	0,1	0,1	0,1	1,0
20	s/1	F_3	0,1	0,1	0,1	1,0

To make the network self-checking, it is only necessary to modify the subnetwork which generates line 20. Specifically, lines 14 and 15 could be fed into a separate NAND gate so that line 20 no longer fans out. This is shown in Figure 3.7. Since the analysis of the network still applies for all other lines and the network has been shown self-checking with respect to those lines, only the part of the network modified will be considered, i.e., lines 2,9,14,15,20,23,25,*1,*2, and *3. Using Algorithm 3.1 and analyzing the only output affected by the change, output F_2:

1. Equivalent lines are (20,23) and (25,*3). So the reduced set of lines to analyze is: 2,9,14,15,25,*1,*2.

2. Lines satisfying condition A of the algorithm are: 2,15,*1. This leaves lines: 9,14,25,*2,*3.

3. Lines satisfying condition B of the algorithm are: 14,20,*2,*3. Only line 9 remains.

4. Line 9 satisfies the multiple output condition of the algorithm as before and so the network is completely self-checking.

In this example of seventeen gates and three inputs a simulation may be about as fast as the analytic approach used. However, for larger networks considerable calculation can be saved by using the analytic approach. It may be desirable to combine the analytic approach and a simulation to analyze the lines used by more than one output. The analytic approach has the added advantage of giving more insight into what needs to be done to make the network self-checking.

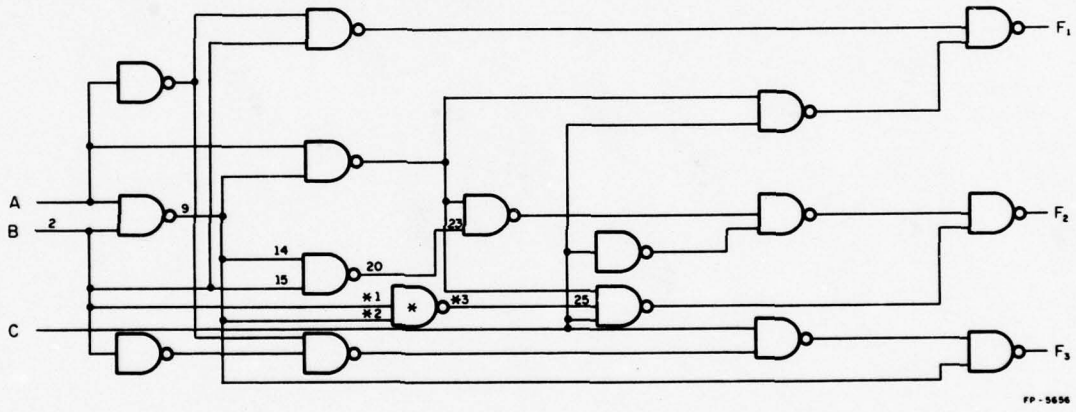


Figure 3.7. Self-checking modified network of Figure 3.4

4. SEQUENTIAL SCAL

4.1. Introduction

Only combinational networks have been considered in previous chapters. However, most systems used in the world are sequential in nature. Reynolds [REYN1] has presented a design technique for sequential machines which will be briefly summarized in Section 4.2 for background. A memory-efficient approach to sequential machine design is presented in Section 4.3. This utilizes the technique of code conversion. In Section 4.4 a technique for direct implementation of sequential machines from the state table is presented. Finally, in Section 4.5 an example of the methods of Sections 4.2 and 4.3 will be given for comparative purposes.

In the discussion the standard sequential machine model in Figure 4.1a will be used. It is modified for alternating logic as shown in Figure 4.1b. The asterisk indicates that the values are encoded in some manner, specifically including signal alternation. The objective of this chapter is to develop SCAL sequential machines which are simple to design and/or at minimal cost.

4.2. Dual Flip-Flop Implementation

Reynolds [REYN1] has discussed at length the details of the dual flip-flop implementation approach. Only a brief summary will be presented here.

Only two steps are needed to convert a sequential machine, as the one in Figure 4.1, to an alternating logic sequential machine. First, the

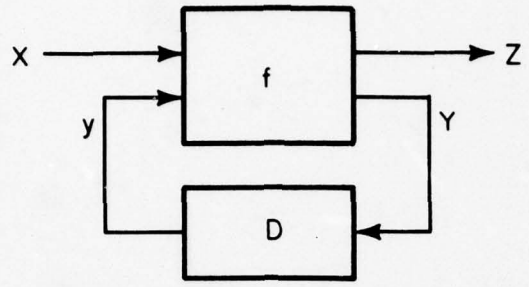


Figure 4.1a. Standard sequential machine model

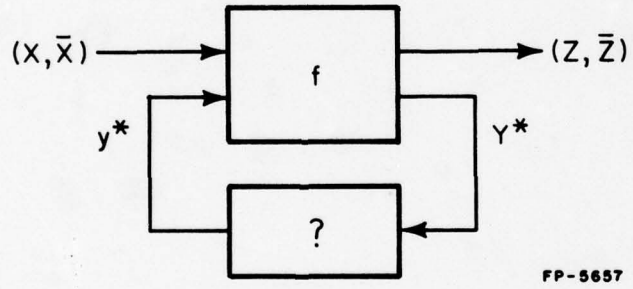


Figure 4.1b. Alternating logic sequential machine model

combinational network should implement a self-dual function. At most, this requires the addition of one extra variable--specifically the clock line. Second, in the dual flip-flop implementation, the number of delays in the feedback path is doubled. As the inputs alternate in value, the output and feedback variables alternate. This is shown in Figure 4.2a.

The value of y is two time units delayed from the value of Y. In a standard sequential machine the inputs to the combinational network are X_t and Y_{t-1} and the outputs are Z_t and Y_t. In the alternating logic implementation in Figure 4.2a the inputs to the combinational logic are X_t and Y_{t-2} in the first time period and X_{t+1} and Y_{t-1} in the second time period, where $Y_{t-2}=\overline{Y}_{t-1}$ and $X_t=\overline{X}_{t+1}$. The outputs are (Z_t,Z_{t+1}) and (Y_t,Y_{t+1}) with $Z_t=\overline{Z}_{t+1}$ and $Y_t=\overline{Y}_{t+1}$ in a correctly operating alternating logic network. An example of a sample data stream is shown in Figure 4.2b.

All input and output signals alternate in unison with the period timing clock. To verify correct operation, it is necessary to monitor not only the Z outputs, but also the Y outputs to ascertain that the correct state information is fed back to the network. The method of checking the lines is critical to the operation of a SCAL system and will be discussed in the next chapter.

4.3. Code Conversion Technique

In detecting faults of a given fault class in a system, different encodings of the data may be used for different parts of the system as long as the required code space distance is maintained in the codes. This requires code conversion between parts of the system. In the case of SCAL, single

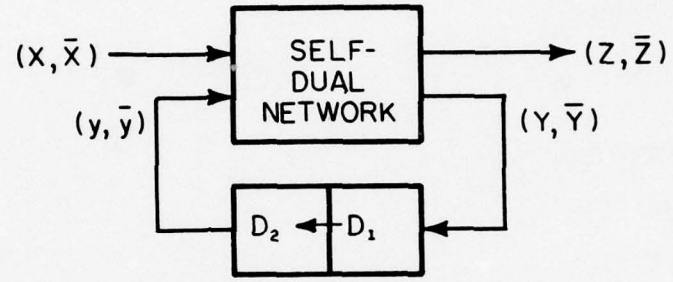


Figure 4.2a. Dual flip-flop sequential machine

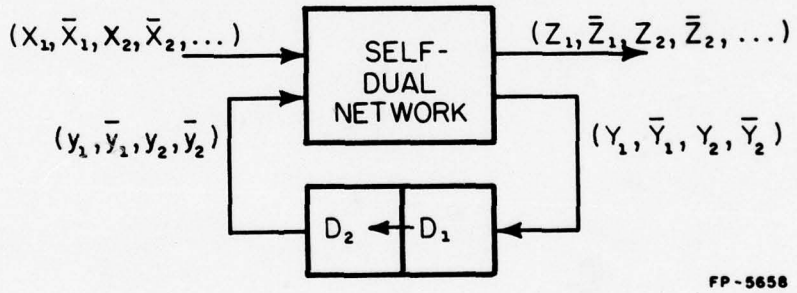


Figure 4.2b. Sample data input stream

fault detection requires a code space distance of two. Therefore in the space domain a single parity bit provides sufficient code space distance to maintain detection of a single fault.

This conversion characteristic is useful to match the failure mode of a particular subsystem with an appropriate code for detecting the most likely faults. For example, one approach to storing alternating signals is to double the size of memory, as was shown in the previous section. During successive time periods the alternating signals are obtained for use by the processor. For an n-bit word of a normal system this would require 2n bits for storing the word. This method is inefficient in using memory since only n+1 bits are required to provide the necessary code distance for single fault detection. Therefore, an approach which translates the coded information to and from a more efficient encoding in memory is desirable.

Figure 4.3 shows a system level representation of an approach using translation between time and space encoding for data storage. The feedback variables from the conventional logic $(Y,\overline{Y})$ are translated to a code T which is stored and retrieved later as code t. This is converted to give the same $(y,\overline{y})$ input to the combinational logic as the dual flip-flop approach required for alternating logic. In this manner an efficient match of the qualities of alternating logic for reduced processor costs is combined with an efficient match of parity encoding to minimize memory costs. Without loss of generality, an even word size, n, and even parity will be used in further analysis.

A system wide clock will be used to control in which of the two time periods the alternating logic system is operating. It will be represented as

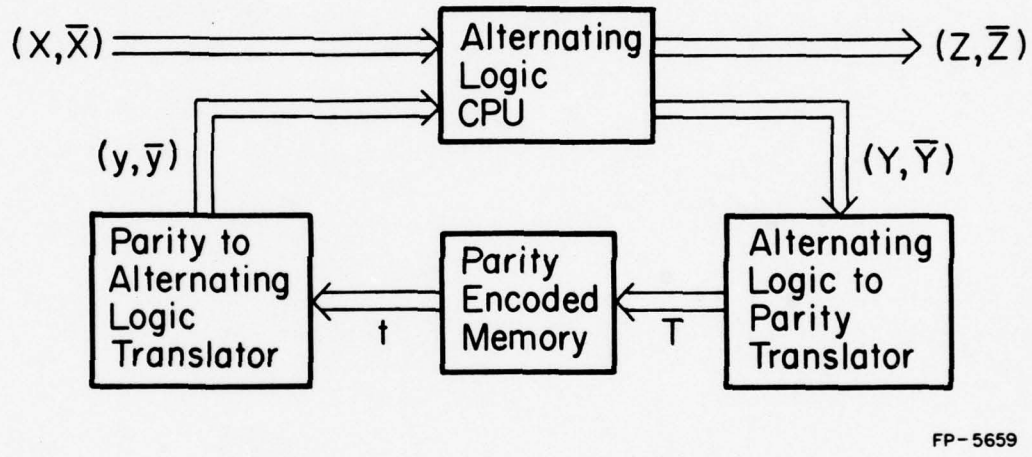


Figure 4.3. System representation of code translation

$(\phi,\bar{\phi})$ and will have the alternating value pair of (0,1). The normal system clock operates at twice the frequency of $(\phi,\bar{\phi})$ and is used to generate $(\phi,\bar{\phi})$. Positive edge-triggerred D-type flip-flops will be used, so that data are latched on the 0 to 1 transition of their inputs. The period clock could be used as an additional input, when it is necessary to convert an odd word size to even word size or to change the parity.

As can be seen in Figures 4.4a and 4.4b, the translators are simple and inexpensive. The alternating logic to parity translator will be referred to as ALPT and the parity to alternating logic translator will be referred to as PALT.

The complete system is shown in Figure 4.5. The n outputs of the combinational network $(Y,\bar{Y})$ are input to the ALPT where the Y value is latched on the 0 to 1 transition of ϕ and the parity of $\Sigma\bar{Y}_i$ is latchedon the 1 to 0 transition of ϕ. These n+1 values are stored in memory and when retrieved an XOR with $(\phi,\bar{\phi})$ or (0,1) is taken with each line to form $(y,\bar{y})$. The complemented parity of all the y outputs is latched on the 0 to 1 transition of ϕ. It is output with the $\Sigma\bar{Y}_i$ parity calculated earlier to give a 1-out-of-2 code to provide the self-checking property of the system. If the feedback is thru only one level of memory, then no additional latches would be required for memory, since the ALPT stores one time period of information in its latches.

If random access memory is used, then the address selection of memory must be self-checking. This is accomplished as discussed by Dussault [DUSS1] by including the parity of the address with the parity of the data stored in the ALPT. Upon retrieval the same is done in the PALT and, if correct, a

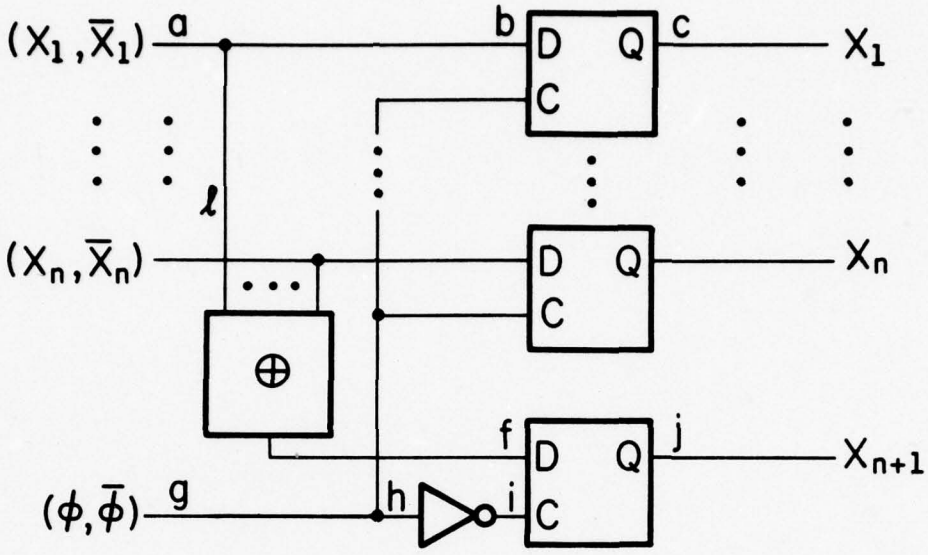


Figure 4.4a. Alternating Logic to Parity Translator (ALPT)

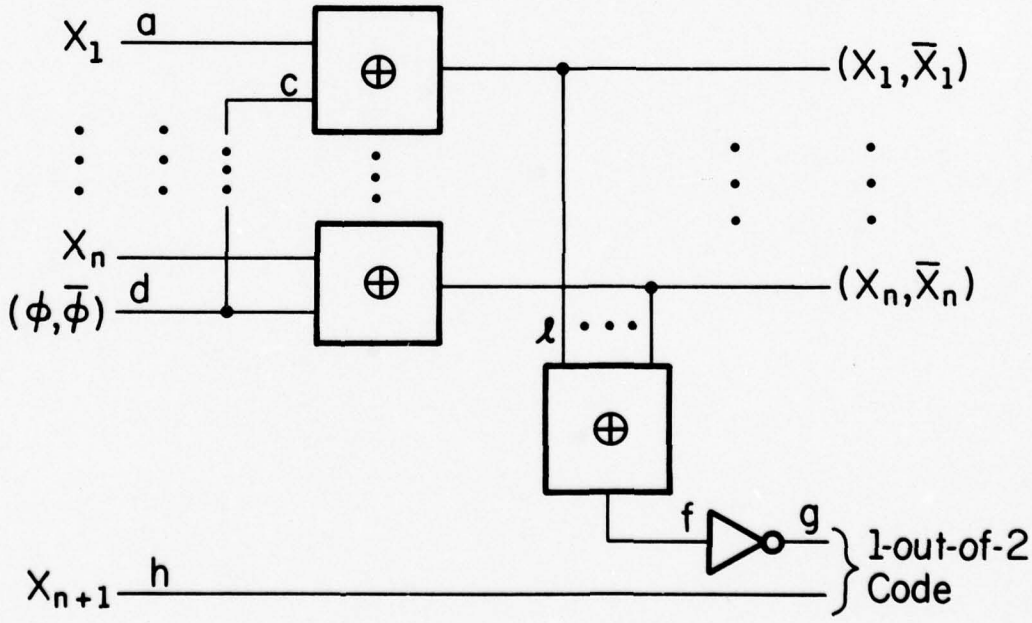


Figure 4.4b. Parity to Alternating Logic Translator (PALT)

FP-5660

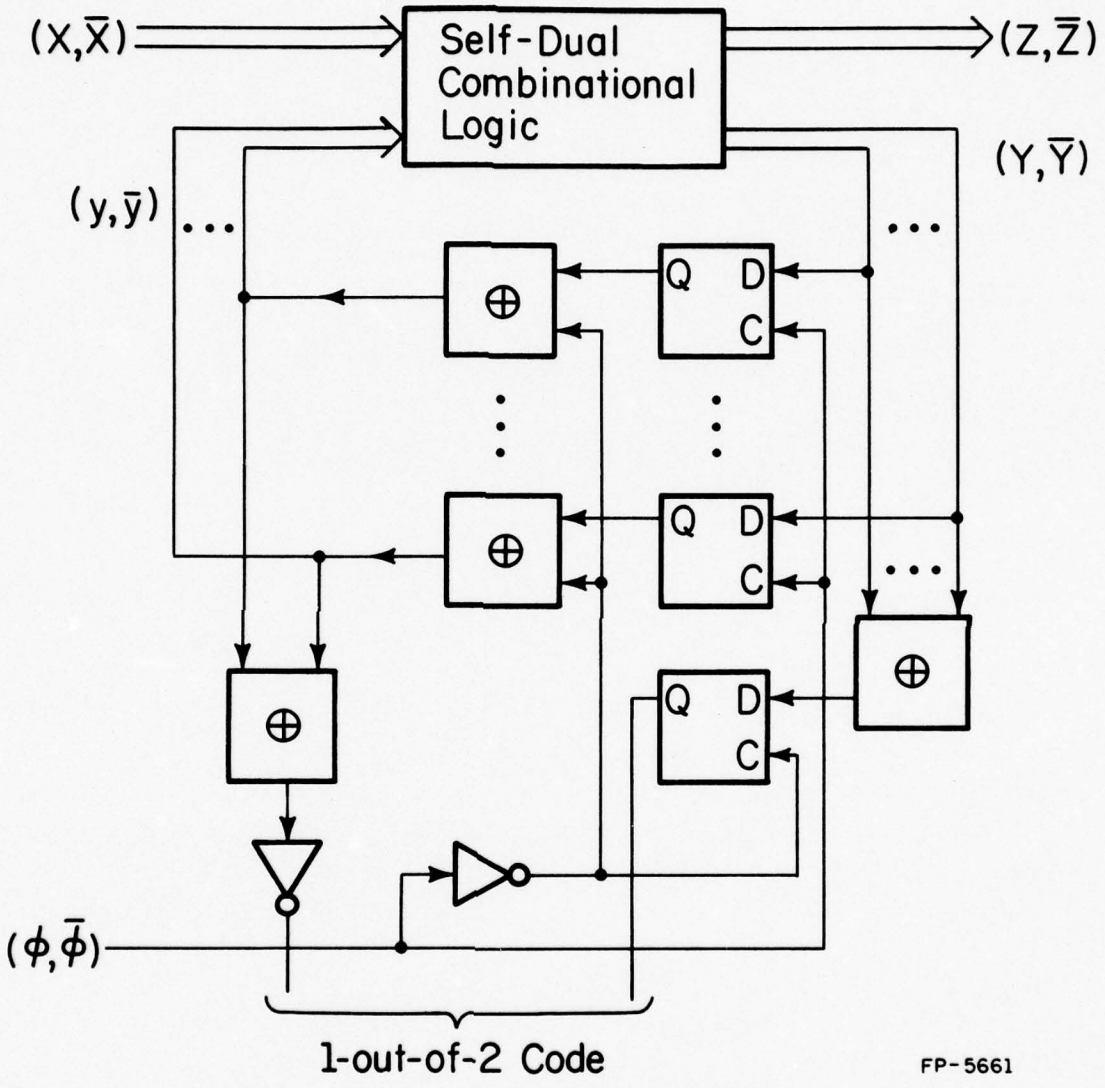


Figure 4.5. Complete code translation system

1-out-of-2 code is generated. For example, if the parity of the data is odd and the parity of the address is even, then the output parity line of the ALPT will be odd. Similarly, when the data are retrieved, the parity of the data out of memory and the parity of the address are combined to give odd parity. The PALT will then output a 1-out-of-2 code. However, if a failure occurred on either write or read so one line of address is incorrect, then a 1-out-of-2 code would not be generated. The system can be shown to be self-checking [DUSS2].

For the system to remain self-checking, all the units in the feedback path must be self-checking. The determination of why the translators work and why they are self-checking will now be presented.

The feedback outputs must be checked along with the external outputs of the combinational logic by a self-checking checker. If the feedback variables were not checked, a single fault in the logic generating the feedback variables could go unnoticed while causing several outputs to be invalid. This violates the fault secure property required in a self-checking system. Further analysis is given in Chapter 5.

Therefore, a single fault on one of the combinational logic feedback output lines can be assumed independent of a failure on any other line. Furthermore, except for X_{n+1}, each output line of the ALPT is separately generated from only one feedback line and so a single fault there also affects only one line. Now consider the self-checking characteristic of the ALPT.

Theorem 4.1. The ALPT is self-checking if the parity of its output is checked.

Proof: Each class of lines will be considered separately, using representatives of the classes as in Figure 4.4a.

1. A stuck-at s fault on lines b,c,e,f, or i will be detected whenever the value of the line is $\bar{s}$, since the parity of the output will be incorrect. Each such input is a test and no input generates an incorrect parity output. Therefore, the ALPT is self-checking with respect to b,c,e,f, and i.

2. A fault on line d will be detected whenever the values of the lines b and g change since the value of line c will not be changed to the new value of line b as it should when the clock changes. Each such input is a test and no input generates an incorrect parity output. Therefore, the ALPT is self-checking with respect to line d.

3. A fault on lines h or j will be detected whenever the value of the lines f and g change since the value of line i will not be changed to the new value of line f as it should when the clock changes. Each such input is a test and no input generates an incorrect parity output. Therefore, the ALPT is self-checking with respect to lines h and j.

4. A stuck-at s fault on line a when a should be $(\bar{s},s)$ will cause output X_1 to be the opposite of what it should be. However, X_{n+1}, will be the same as it would be without the fault since it is determined in the second time period when a should have been s. Alternatively, if a should be $(s,\bar{s})$, then output X_1 will be correct but X_{n+1} will be incorrect. In either case, the parity will be incorrect and the failure detected. All inputs will cause the parity to be incorrect and so the ALPT is self-checking with respect to line a.

5. If g is stuck, then the latches will retain a constant output which may not be correct, but the parity will remain correct. If this happens, the network is not fault secure. However, if it is assumed that all fan out of the clock ϕ is from a common node, then only one clock input line or all clock lines must fail, i.e., not some subset of clock lines. The case of one clock line failing has been considered in steps 2 and 3 above. If all clock lines fail, then the system will stop and no output, correct or incorrect, will be generated. Thus it is fault secure. Any input which is applied while ϕ has failed will cause this and so it is self-testing if the system shut-down is regarded as a noncode state. So the network is self-checking with respect to g.

Since the network is self-checking with respect to all lines in the network, the network is self-checking. Q.E.D.

Three conditions for the network to be self-checking were assumed in the proof: (1) all fan out of clock line ϕ is from a common node, (2) stopping the system is regarded as a valid self-checking operation in event of failure (this will be discussed further in Chapter 5), and (3) the output parity must be checked.

Theorem 4.2. The memory is self-checking if its output parity is checked.
Proof: In the memory all outputs are independent of each other and so any single fault affects at most one output line. If the output line is sensitized, then the fault is detected. Some input will sensitize the memory line or the memory unit is unnecessary and may be removed. Since the memory is self-testing and fault secure for any fault, it is

self-checking. Q.E.D.

The only remaining unit to be considered in the code translation process is the PALT.

Theorem 4.3. The PALT is self-checking if its 1-out-of-2 code output is checked.

Proof: The analysis will be done on classes of lines labeled in Figure 4.4b.

1. A fault on lines e,f,g, or h is detected by the 1-out-of-2 code checker used. These are fault secure and self-tested when sensitized and so the network is self-checking with respect to e,f,g, and h.

2. A fault on line b causes b not to alternate which is detected when sensitized in the combinational unit receiving the non-alternating value. It is fault secure and self-tested when sensitized and so the network is self-checking with respect to b.

3. If line a is stuck-at s and is sensitized, then b will be $(s,\bar{s})$ when it should be $(\bar{s},s)$; i.e., it alternates incorrectly. However, the parity on line f would be the opposite of what it should be and so f and h would be opposite and g and h would be the same. This would be a noncode word and would be detected. So the network is fault secure. Since any input which has line a at $\bar{s}$ is a test, the network is self-checking with respect to line a.

4. A fault on lines c or d would cause b not to alternate and the analysis of step 2 applies to show the network self-checking with respect to c and d. Since the network is self-checking with respect to all lines in the network, the network is self-checking. Q.E.D.

Theorem 4.4. Combining an ALPT, memory, and a PALT provides a self-checking feedback.

Proof: By Theorems 4.1, 4.2, and 4.3 each subunit is self-checking, provided the conditions of the Theorems are all met. These are (1) the 1-out-of-2 code of the PALT must be fed to the system checker, and (2) the parity of the ALPT output data lines must be checked. Condition 1 is automatically satisfied in any self-checking system which must have a self-checking checker and appropriate hardware, discussed in Chapter 5. Condition 2 is satisfied since the parity is checked in the PALT by (1) determining the complement of the parity of the data lines and (2) feeding that with the parity calculated by the ALPT (line X_{n+1}) to a checker. Therefore, the conditions are sufficient for the feedback system to be self-checking. Q.E.D.

Thus a simple method is provided to build a sequential SCAL at minimal memory cost.

An alternative self-checking feedback network which provides the code space distance necessary is shown in Figure 4.6. This combination also provides a reduced checker cost for the system since the first checker used could be the same as the one required to monitor the feedback variables. The second checker would not require the logic used only in the generation of the f output. The translators are the same as before except that the parity generator and checking circuitry are not used.

4.4. Direct Implementation

The two previous sections have considered methods of converting an arbitrary sequential system. This section will discuss methods of directly

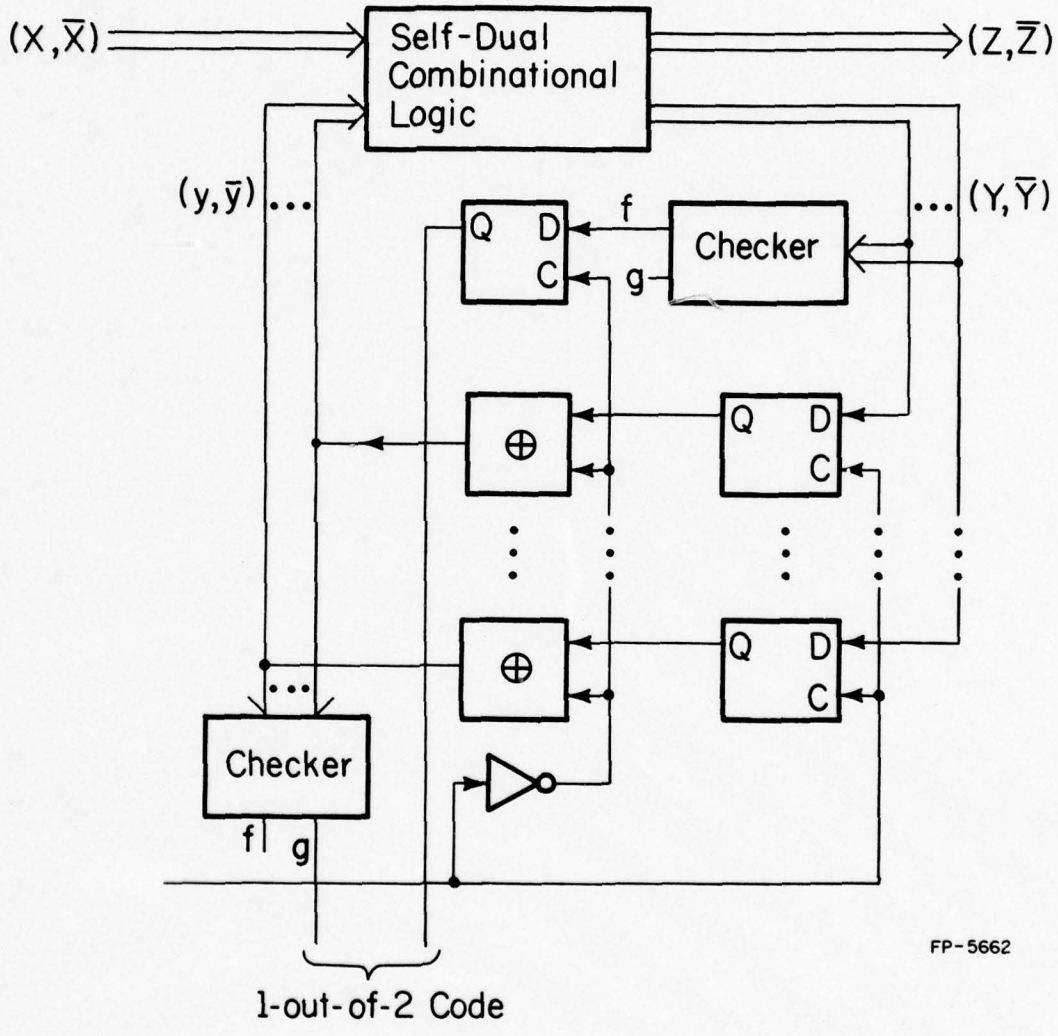


Figure 4.6. Modified self-checking feedback network

implementing a sequential SCAL system from the logic description. As such, it will not have the fixed structural properties of the previous methods and so will be more difficult to design, but may be less expensive in hardware.

The possible approaches to designing the feedback logic for a SCAL system are enumerated in Figure 4.7. A checker is normally used on the feedback variables. If it were not used and if a fault occurred which changed the value of one of the feedback variables, then the input to the combinational logic would receive a noncode input two time periods later. If the fault forces one of the system outputs to a noncode value, then the fault is detected. However, if the fault only affects the feedback variable which originally had the fault, it could allow the feedback variable to generate an incorrect code word. This would cause the system to be in the wrong state and would violate the fault secure condition of self-checking systems.

Another possibility is that the noncode feedback causes another unchecked feedback variable to generate a noncode output. With more than one feedback variable generating noncode outputs fed back into the combinational logic, it would appear as a multiple fault to the combinational logic. This is not included in the self-checking fault coverage and, consequently, the system would not be self-checking.

Appropriate analysis of the network could be made or design strategies developed which would insure the fault was detected before it incorrectly affected the output. These would be fairly complex and the additional hardware logic cost would probably be greater than the excess checker cost. Analysis of the logic could save some of the checker cost if certain feedback outputs did not need to be checked. However, in further analysis it will be

Output Checker Used

Input:	Output: Parity	Output: Alternating
Parity	1	2
Alternating	3	4

No Output Checker Used

Input:	Output: Parity	Output: Alternating
Parity	5	6
Alternating	7	8

FP-5663

Figure 4.7. Feedback logic design in SCAL

assumed that an output checker is used.

Case 4, alternating input and alternating output, was discussed in Sections 4.2 and 4.3. The other cases use a combination of space and time redundancy directly and are candidates for direct implementation from the logic descriptions. Although parity code will be discussed, any space redundant code could be considered. Other codes may have lower system cost by reducing the cost of combinational hardware. However, the standard approach to sequential machine design has been to assume a high memory cost. So overall cost is lowest if memory cost is lowest. Since parity only requires one extra feedback variable, and some redundancy is required to provide self-checking, it is the least expensive approach.

Case 1, parity input and parity output, is effectively what is used for feedback in the code conversion technique if the ALPT and PALT are regarded as part of the combinational logic. However, if the parity is used by the combinational logic directly as input and generated directly for the output, the costs of the ALPT and PALT could be saved. This completely loses the advantage of alternating logic. Although it may be used in space redundancy approaches, it will not be considered further here since it has the cost of double time requirements of SCAL without any compensating value.

Case 2, parity input and alternating output, may be implemented by using the ALPT to convert the alternating feedback to parity. There is little reason to use this approach since it loses the advantages of alternating logic in combinational logic without reducing memory costs.

Case 3, alternating input and parity output, may be implemented by having the combinational logic generate a parity code output and using a PALT module.

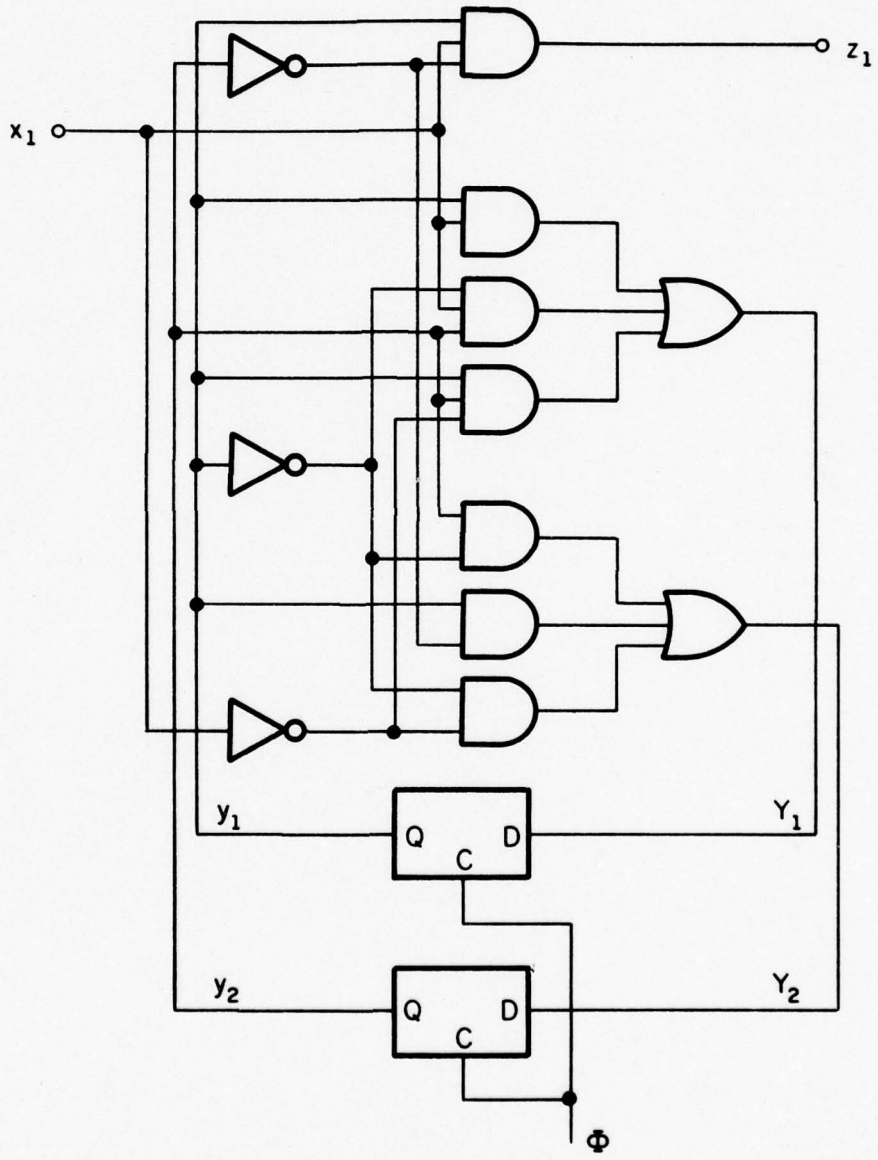
To generate the parity code either the combinational outputs during the first or second time period could be used. Whichever outputs are used, the parity code must be generated independently of the other outputs so that a fault to an output will not also change the parity. Similarly, using a parity code requires that an even number of outputs not be changed at the same time so logic sharing is greatly restricted. However, if two outputs are latched during opposite periods, then they may share logic. A simple example of this was the ALPT which took the parity of the outputs in the opposite time period in which they were latched. The ALPT is also the least expensive way of generating the parity.

Cases 1,2, and 3, which directly use parity code in the combinational logic, show little promise in application to alternating logic. The code translation approach of Section 4.3 is easier to design, more structured, and almost always less expensive. Therefore, techniques of directly implementing sequential SCAL through modified sequential machine design techniques will not be worthwhile.

4.5. Comparative Example of Techniques Presented

An example of the implementation of a sequential machine using the techniques of Sections 4.2 and 4.3 will be given to compare their merits. The direct implementation techniques of the last section are not effective and no example will be given of them.

To enhance the comparison an example from previous work by Kohavi [KOHA] and then Reynolds [REYN1] will be used. It is of a 0101 sequence detector. The original sequential machine of Kohavi is in Figure 4.8. The version



FP-5100

Figure 4.8. Kohavi's 0101 sequence detector

developed by Reynolds is in Figure 4.9 (with ϕ_A a clock operating at twice the frequency of ϕ). The translator implementation is in Figure 4.10. The comparative costs are in Table 4.1, with n and m defined as the Kohavi number of flip-flops and gates, respectively.

As is observed, to achieve self-checking extra logic is required in addition to the double time requirement of SCAL. Assuming flip-flops have a high cost, the cost is least if the translation method is used in implementing sequential machines, and this cost effectiveness becomes even more apparent the larger the machine is. In particular for computers the total memory cost is doubled in the dual flip-flop approach, but is increased very little in the translator approach.

The system-wide checker cost should be divided among each unit in the system; thus the amount of cost to be apportioned for the particular subunit depends on the system size. It is not included in the costs given since checker cost would be a small part of the total system cost. The number of gate inputs and the number of gate delays may also be cost factors to consider.

The general case is also given. It uses the 1.8 cost factor, determined by Reynolds [REYN1], as an approximate cost of converting normal logic to SCAL. Cost factors vary widely from one for an adder to multiples for some logic. The key point is that in the comparison between the dual flip-flop approach of Reynolds and the translator, gate cost is about the same, but flip-flop cost is much less for the translator approach.

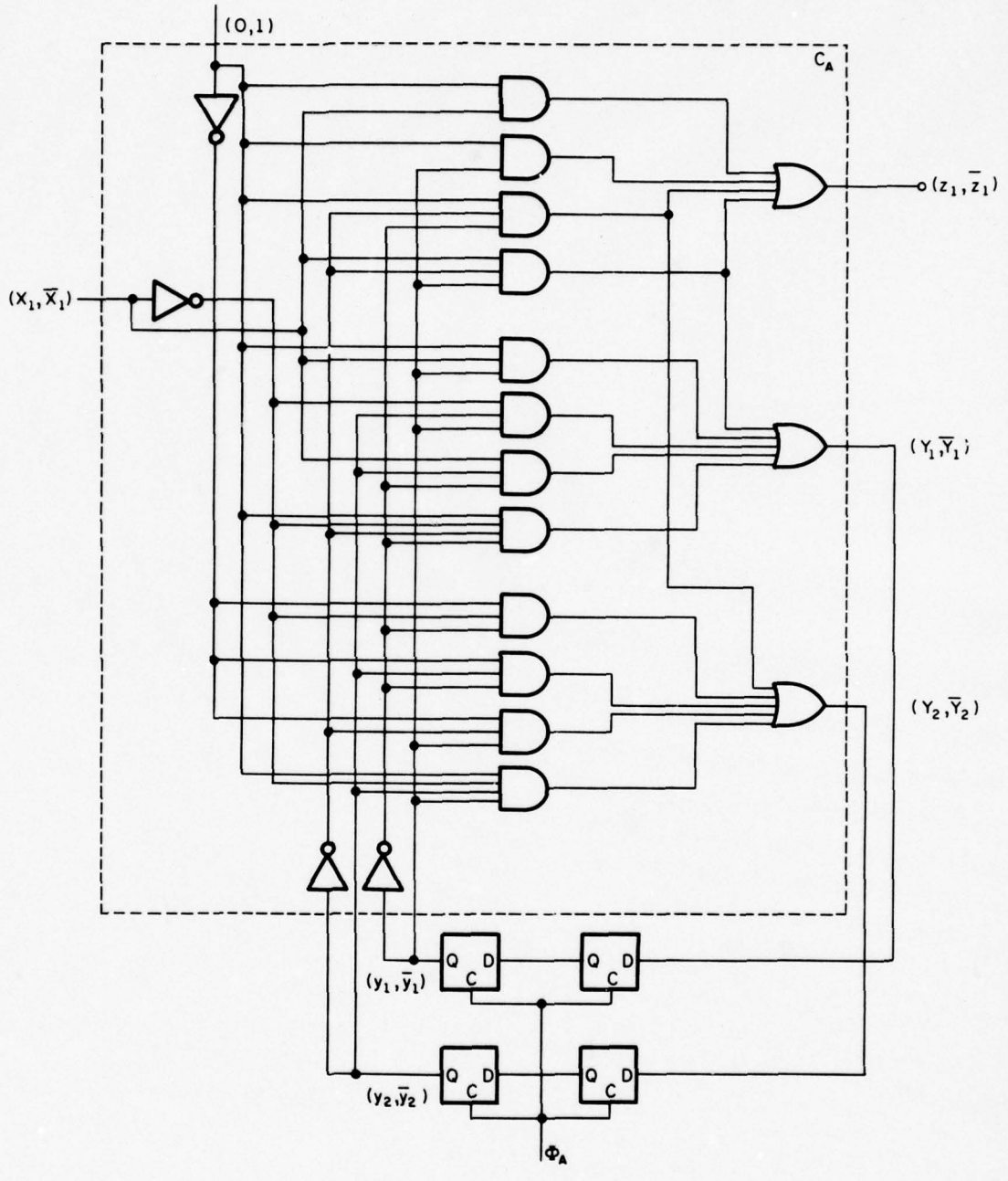


Figure 4.9. Reynold's SCAL 0101 sequence detector

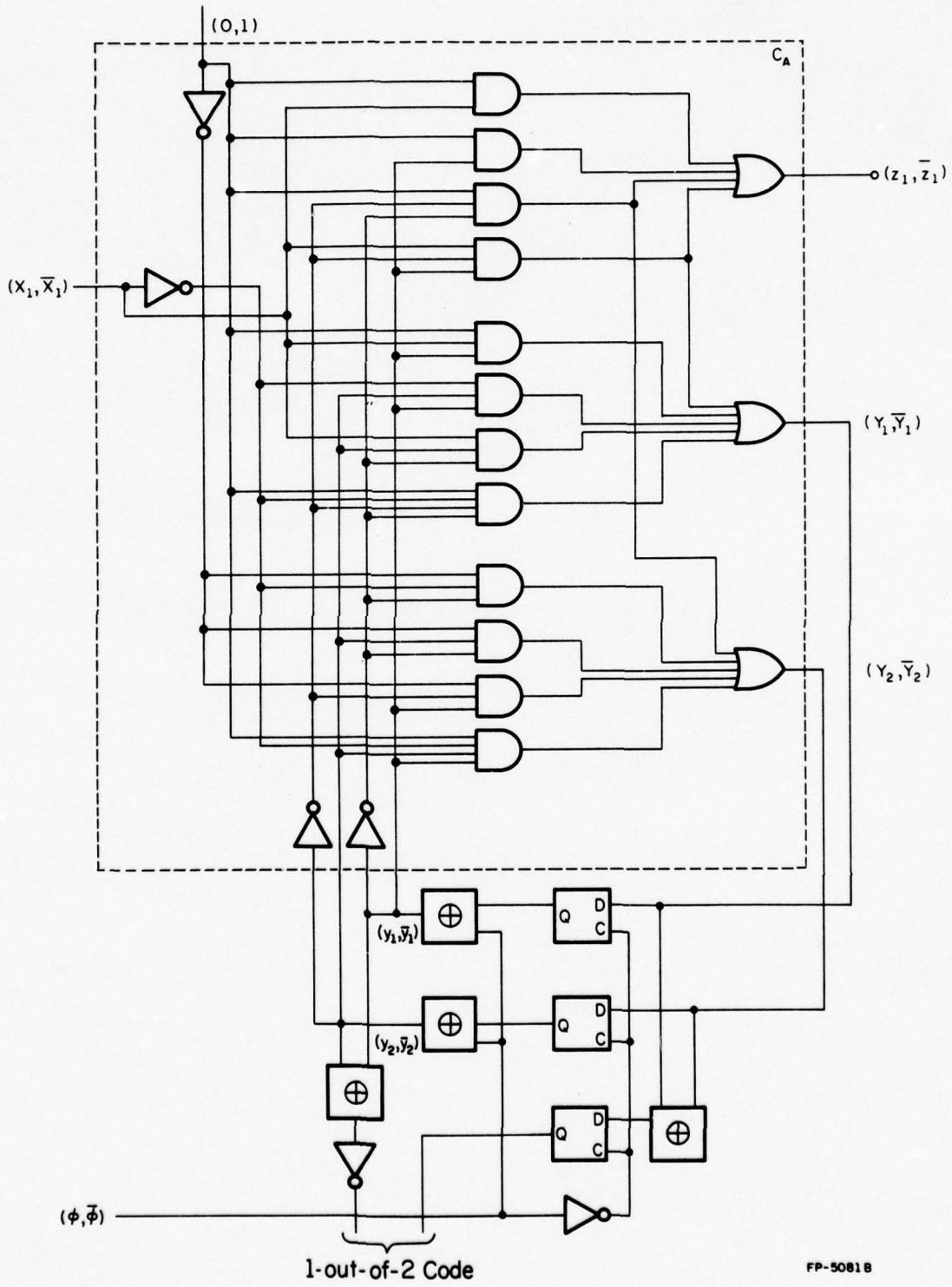


Figure 4.10. Translator implementation of 0101 sequence detector

Table 4.1. Comparative costs of sequence detector

	Flip-Flops	Gates
Kohavi example	2	12
Reynolds example	4	19
Translator example	3	23
Kohavi general	n	m
Reynolds general	2n	1.8m
Translator general	n+1	1.8m+n+2

5. CHECKER DESIGN

5.1. Introduction

This chapter will cover (1) the design of SCAL checkers using the conventional assumption of interdependent outputs, (2) the design of SCAL checkers for independent outputs, (3) the design of SCAL checkers for networks which have some dependent and some independent outputs, and (4) the integration of the checker into the complete SCAL system. In evaluating the checkers it will be assumed there are a multiple number of inputs to the checker (from the network outputs). It is also assumed that a minimum number of checker outputs are desired so that the hardware requirements of the system are minimized. This assumption is substantiated in the last section of this chapter.

5.2. Dual-Rail Checkers

The conventional approach to designing checkers has allowed the inputs to be interdependent; i.e., the outputs may utilize some of the same logic in their generation. In the space domain with the classic assumption of unidirectional faults, the network is usually constructed with inverter-free logic beyond the input level. In this case the interdependency does not imply any extra constraints on the checker design.

In the time domain, Reynolds [REYN1] proposed using a set of flip-flops to record the network output in the first time period and feeding these delayed outputs and the second time period outputs into a dual-rail totally self-checking checker (TSCC) [ANDE1]. This entire checker of Reynolds will be

If the module is assumed to not have a fault within it, then the internal state of (c,f,g) would never reach (1,1,1) in the analysis above. As was shown before, the connections to the module allow the system to be self-checking if the module is assumed fault-free.

Theorem 5.2 means that no completely self-checking system can be built with the normal logic gates available. There must be some nonstandard unit which is used as the hardcore for the system, or the module that is not self-checking must be replicated to obtain the desired reliability. This does not consider some other mode of operation such as transition oriented logic or pulse-mode logic which probably have other untestable faults.

One other possibility for providing self-checking capability in the system is to feed back the checker outputs so they may be latched during the next clock period, as shown in Figure 5.7. Once a faulty output is signalled by the checker it will then remain at that noncode word, (0,0) or (1,1). Presumably this status is displayed and the fault recognized by the operator. All operations since the last time the checker output was checked may be in error.

System-wide all the checkers in the system can be fed to one final checker without loss of the self-checking characteristic. Only this one checker needs to be monitored in assuring self-checking operation.

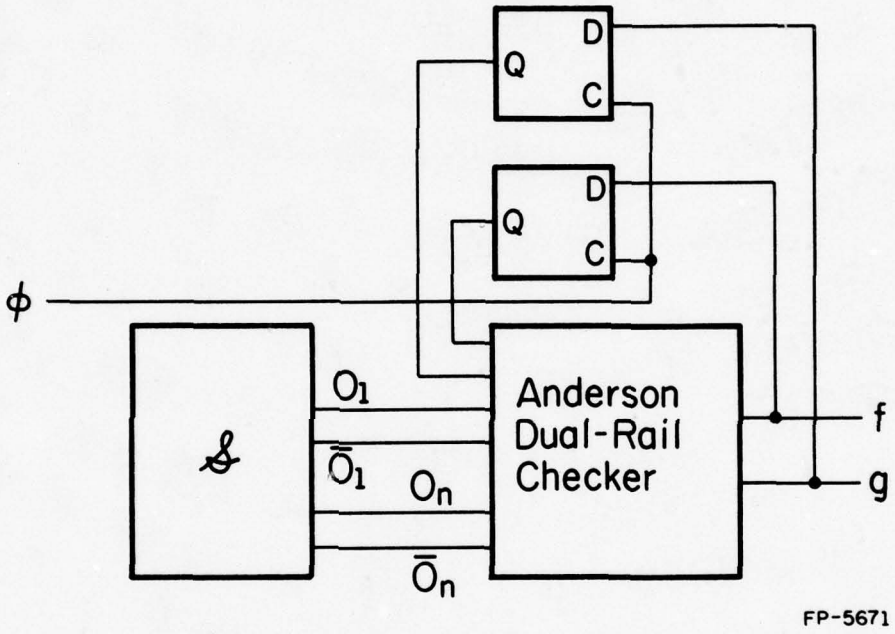


Figure 5.7. Feedback of checker outputs

referred to as a dual-rail checker and is shown in Figure 5.1a. This checker can use the first stage of the dual flip-flops used in the dual flip-flop sequential logic design as the flip-flops for the feedback lines being checked. This is shown in Figure 5.1b.

The dual-rail checker requires two outputs, which have been shown by *Anderson [ANDE1] to be the minimum required of a system checked during only* one time period. However, since two time periods are used in the remainder of the alternating system, it may be useful to convert these dual outputs to a single alternating output, as depicted in Figure 5.1c. The output q changes at the rate of the time period clock ϕ , (0,1), and q should be (1,0) if the network is operating properly. It will be (0,1) or constant if there is a fault in the network.

5.3. Independent Line Checker

If the checker inputs are independent, then the requirements of the checker design are reduced. Reynolds [REYN1] proposed using an exclusive-OR (XOR) gate network as a checker.

Theorem 5.1. An XOR network is a self-checking single output checker for SCAL if each XOR gate has an odd number of inputs.

Proof: If an odd number of inputs feed each XOR gate and each input alternates in value with the period clock, then the output will also alternate in value. This applies to all of the XOR gates in the network, so all the lines alternate. By Theorem 3.6, a network is self-checking with respect to the alternating lines in the network. Therefore, the network is

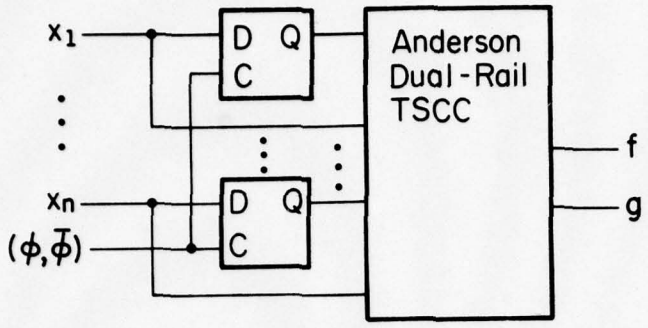


Figure 5.1a. Reynold's dependent line dual-rail self-checking checker

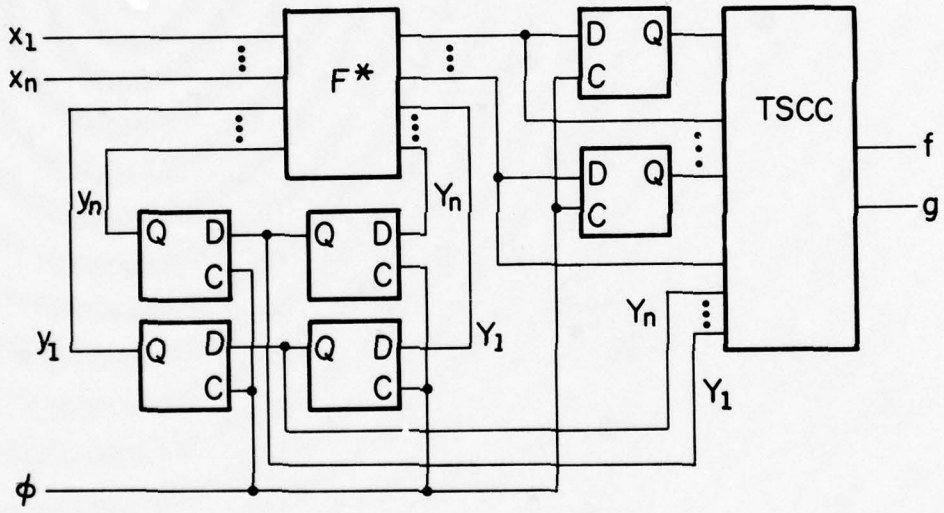
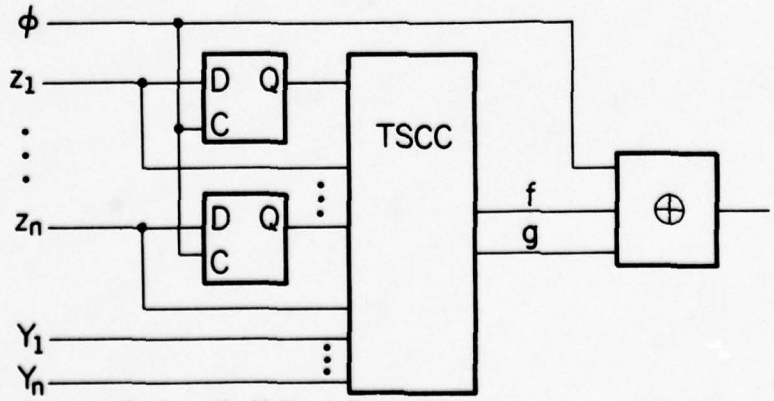


Figure 5.1b. Reynold's checker in dual flip-flop network



FP-5665

Figure 5.1c. Modified Reynold's checker with alternating output

self-checking. Q.E.D.

An example of an XOR self-checking checker is given in Figure 5.2a. Note that the alternating period clock is added to the last XOR gate so that all the XOR gates have an odd number of inputs.

The gate cost can be made minimal by proper construction of the XOR tree. One output is the minimum possible so the network provides the minimum cost checker. To get a two-rail output, it is only necessary to add a flip-flop on the output for the first time period, as in Figure 5.2b. The checker output is monitored in the second time period. It is also possible to build a self-checking checker network with even input XOR gates as shown in Figure 5.2c. In this case, only the output pair (0,1) is a code word and all other output pairs are noncode words. This type of checker is less cost-effective than the checker composed of odd input XOR gates.

5.4. Mixed Checker Designs

As was stated before, the requirement for a network to be able to use an XOR checker is that its outputs be independent. This restriction may be relaxed provided the outputs satisfy certain constraints similar to those tested in Chapter 3 for the network to be self-checking.

Specifically, if an even number of the network outputs brought into the checker are stuck, then the parity would be unchanged and so the checker would incorrectly give a code word output. However, if an odd number of the network outputs are stuck, then even if another network output should alternate incorrectly the fault would be detected. A summary is provided in Table 5.1 of the conditions for up to three network output faults in which the XOR

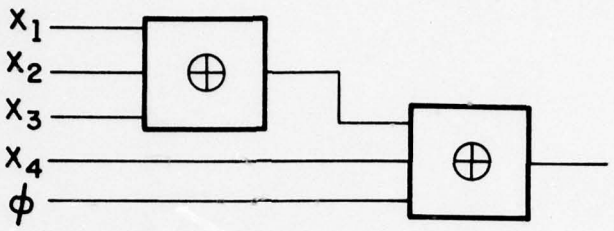


Figure 5.2a. XOR self-checking checker

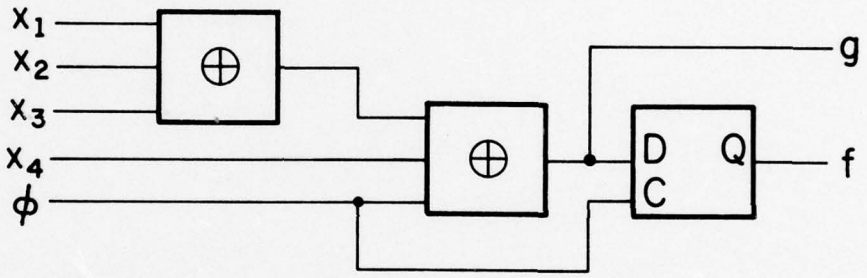


Figure 5.2b. Dual-rail output XOR checker

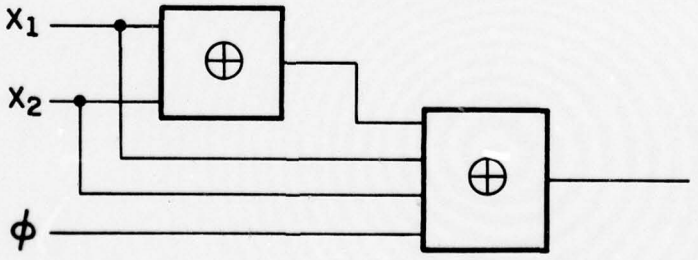


Figure 5.2c. Even input XOR checker

Table 5.1. Conditions where XOR checker suffices

Number Inputs Stuck	Number Inputs Alternating Incorrectly	Result	Checker Operation Proper
0	0	Proper Operation	Yes
0	1	Fault Not Detected*	Yes
1	0	Fault Detected	Yes
0	2	Fault Not Detected*	Yes
1	1	Fault Detected	Yes
2	0	Fault Not Detected	No
0	3	Fault Not Detected*	Yes
1	2	Fault Detected	Yes
2	1	Fault Not Detected	No
3	0	Fault Detected	Yes

checker will suffice. A self-checking network will not output an incorrect alternating word without at least one output not alternating. Therefore, the checker does not need to check for this failure. These cases are marked by an asterisk in Table 5.1.

Using the fact that the XOR checker will fail only when an even number of the network outputs being checked have failed to a constant value, conditions similar to those in Chapter 3 for determining whether a combinational network is self-checking can be developed to determine whether the XOR checker is self-checking. All lines which have a fan out path to more than one output must be considered. This includes the inputs. Where there is a large amount of multiple use of some logic it is fairly likely that the network will not satisfy this condition. Since the inputs feed more than one output network, the way in which the function is implemented may require drastic alteration in order not to allow generation of any pair of nonalternating outputs in the event of failure. In order to modify the network so that it would satisfy this condition, there is a high probability that more logic would be required than adding the additional logic to the checker which allows the checker inputs to be interrelated. So, this is not a useful approach to pursue further.

However, in many networks some of the outputs may use common logic while others do not. Furthermore, it is possible to partition the network according to the subnetworks used by groups of outputs. In this case, a checker design utilizing the mix of dependent and independent variables is appropriate.

Algorithm 5.1:

1. Put all variables which are independent of all other variables in partition A. The remainder are in partition B.

2. Further divide the outputs in B into subpartitions B_i so that the outputs form groups in which each output in the group is not dependent on logic used by any output outside the group.

3. Among each B_i, one of the outputs may be placed in the A partition as long as it does not alternate incorrectly for any fault. A network modification like those discussed previously could change the way the partition is made and hence reduce the checker cost by placing more of the outputs in the A partition.

4. All the outputs in the A partition may be checked with the XOR networks. All the outputs remaining in a B partition must be checked by the dual-rail input checker for dependent inputs.

Notice that in the example in Figure 3.7, all the outputs would initially be in the same B partition. Further, they would all be in the same B_i, partition B_1. Either output F_1 or F_3 could then be put into the A partition. However, they could not be both in the A partition. F_2 must remain in the B_1 partition since it does alternate incorrectly for faults on lines 9, 19, and 20 in the network.

Consider also the following example. Suppose there are nine output lines from a function. Outputs 1, 2, 3 are independent of any other output. The groups of outputs which share logic are (4, 5, 6), (6,7). amd (8,9). Outputs 5 and 8 generate an incorrect alternating output for a line they share with other outputs in their groups. The dual-rail checker implementation is in Figure 5.3a. The algorithm proceeds as follows:

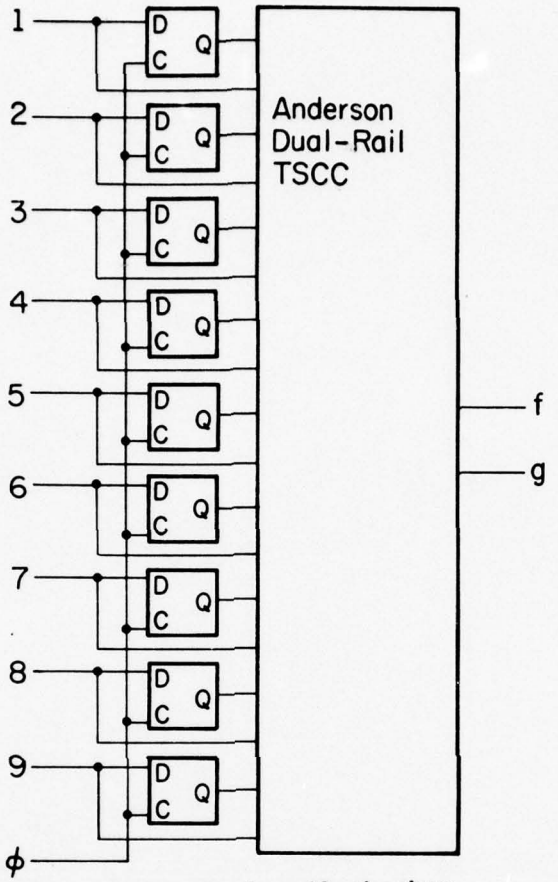
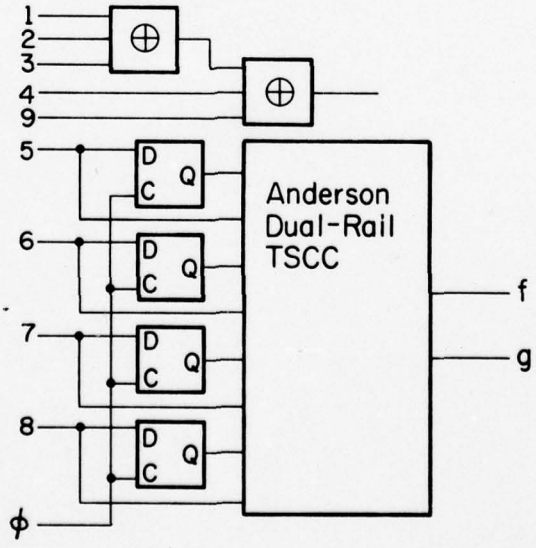


Figure 5.3a. Dual-rail checker



FP-5667

Figure 5.3b. Minimal checker implementation of example

1. A=[1,2,3]; B=[4,5,6,7,8,9]

2. B_1=[4,5,6,7]; B_2=[8,9]

3. A=[1,2,3,4,9]; B_1=[5,6,7]; B_2=[8]

4. Outputs 1,2,3,4,9 are checked by an XOR checker. Outputs 5,6,7,8 are checked by the dual-rail checker. The implementation is shown in Figure 5.3b.

The checker outputs of the two checkers need to be combined in order to provide a minimum number of checker outputs. There are two approaches: (1) if a single alternating output is desired, then another XOR checker is used with the previous checker outputs as its inputs; (2) if a dual-rail output is desired, then a dual-rail totally self-checking checker is used. These two implementations are chosen for the previous example in Figures 5.4a and 5.4b respectively. In either case, the outputs of the first stage checker which is different from the second stage checker may be incorporated directly into the first stage checker which is the same as the second stage checker; i.e., it is not necessary to have the separate stages.

The cost of using the dual-rail checker only as in Figure 5.3a for an n-line input checker is n flip-flops and (n-1)6 two-input gates for Anderson's [ANDE1] dual-rail TSCC. The case given has nine inputs, so forty-eight gates and nine flip-flops are required. Using the reduced cost checker described, depending on the output checker, either (1) three triple-input XOR gates, eighteen two-input gates and four flip-flops are required or (2) two triple-input XOR gates, twenty-four two-input gates, and four flip-flops are required. Either way, the cost is about one-half of the dual-rail checker's cost for this example. The cost formula for this revised implementation depends on how many outputs are in the A partition. The number of latches in

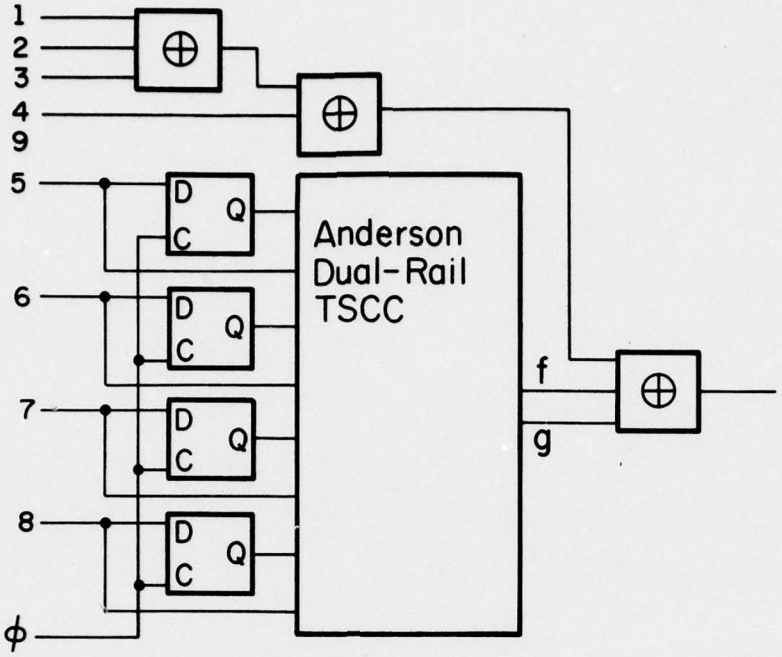


Figure 5.4a. XOR checker output checker

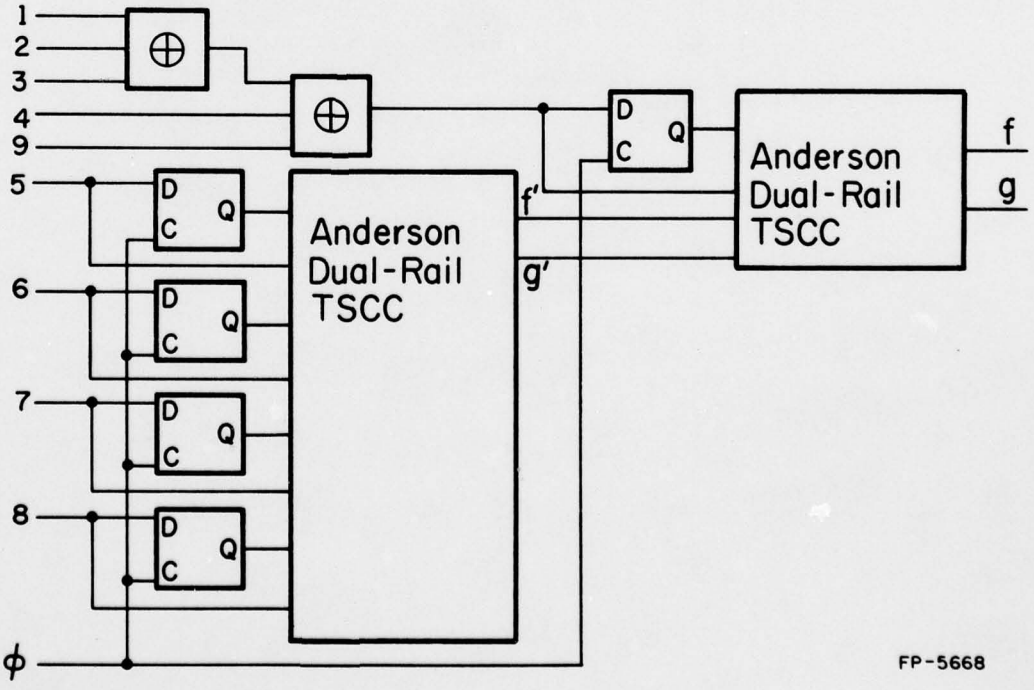


Figure 5.4b. Latching checker output checker

a dual-rail implementation may be reduced in either approach if the lines are also used for feedback outputs.

5.5. Hardcore Elements in SCAL

In order for the checker to have any value for the system, it is necessary to use its outputs to control whether the network continues to operate. Generally, this portion of the network is referred to as the hardcore since it is the part that is assumed not to fail even within the fault model. One approach to hardcore is to physically make the module have a failure rate significantly below the other system components so that it may be regarded as relatively fault-free. Another approach is to redundantly implement the module so that a low probability of all the modules failing is achieved. This first approach is beyond the scope of material covered here; only the second approach will be examined.

In a self-checking system it is desired to terminate operation once a fault occurs which causes an illegal output. It is also desirable to retain the state where the failure occurred. Therefore, turning off power is not acceptable. Either the inputs must stop arriving or, in the synchronous case, the clock must stop. For alternating logic it is assumed a clock is used for synchronization, and so disabling of the clock will be used. Disabling of the input lines (as would be used in an asynchronous system) would be very similar. Particular implementation would vary depending on the network given.

Since the checker output stages have been demonstrated to be convertible, an implementation using either form of checker output is acceptable, although one may cost less. Considered here is the dual output of the latching

checker. To enable the clock only when the output is correct (opposite values), the truth table in Table 5.2 must be implemented. This is implemented using the network module in Figure 5.5a. Regarding the network elements separately, a fault on the XOR gate output to stuck-at 1 will be undetected. Therefore, if the fault occurs, there will be no way of knowing when another fault occurs in the system.

However, if the module is itself regarded as a single gate, then any failure to one of its input or output lines is detected: if f or g is stuck-at 1 or 0, then (f,g) will eventually be (1,1) or (0,0), respectively, forcing the XOR gate output to 0 and turning the clock off; if the clock into or out of the module is stuck-at 1 or 0, then the system will also stop. The failure is detected the moment it is observable, i.e., before any wrong code word is sent out. The fault can then be easily isolated.

Alternatively, if the individual gate faults are to be modeled, then the module can be replicated as in Figure 5.5b. This will decrease the probability of a failure in the hardcore unit being undetected. Whenever a system failure is detected, this unit should be checked or replaced to retain its relatively high reliability. For a probability of failure p and n replications, the probability of hardcore failure becomes p^n. It can be made arbitrarily small for $p<1$.

There is still the question of whether there is any way to build this hardcore unit so that it is self-checking.

Theorem 5.2. There does not exist a self-checking network consisting of normal logic gates (NAND, AND, OR, NOR, XOR, INOR, or NOT gates or flip-flops) which can disable the clock once a fault has been detected.

Table 5.2. Hardcore clock disable truth table

Clock In	f	g	Clock Out
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

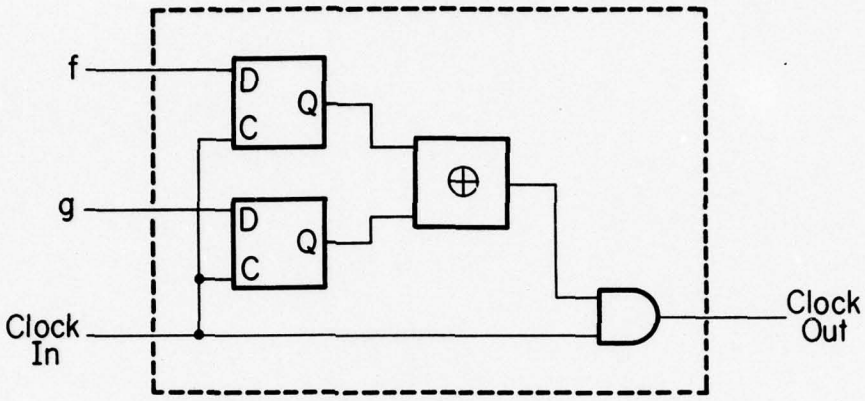


Figure 5.5a. Hardware clock disable module

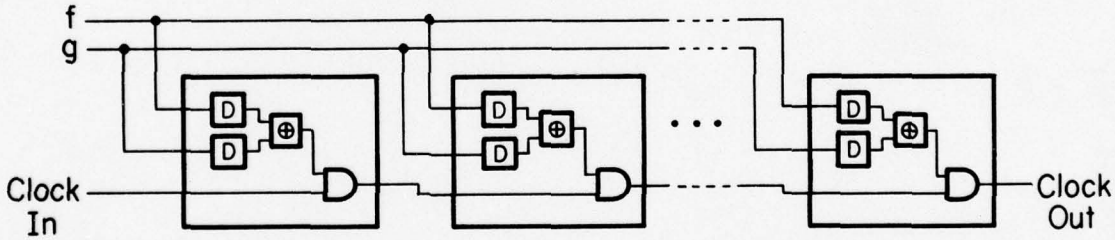


Figure 5.5b. Multiple clock disable modules

FP-5669

Proof: Let the 3-tuple (c,f,g) represent the network's inputs: the clock line and f and g from a dual-rail self-checking checker. The 4-tuple (c,f,g,0) includes the output clock generated. A model of this is shown in Figure 5.6a. The normal transition sequences are shown in Figure 5.6b. This assumes a single input line change at any instant in time. Under correct operation, c should only change when $f \neq g$. By inspection all the transition sequences during normal operation can be observed to be included in Figure 5.6b. Some of the possible sequences upon the occurrence of a fault are shown in Figure 5.6c. The outputs are unspecified, and so are marked as X. Now consider what the value of X must be. It will be shown that no value of X will allow the type of network specified in the theorem to be self-checking.

When the values of f and g indicate a noncode word, the clock output should not change. If it changed values, it would trigger an operation within the system; i.e., the system would not be fault secure. To detect a noncode checker input, it is required that the output remain 0 when (c,f,g) go from (0,1,1) to (1,1,1).

When f fails and (c,f,g) go from (1,1,0) to (1,1,1), the output must remain at 1 to avoid triggering the network to a new state. When c goes to 0 and (c,f,g) go from (1,1,1) to (0,1,1), the output must remain unchanged at 1. However, this now requires that the output for (0,1,1) be 0 when the network operates properly and 1 after a fault occurs. Therefore, the fault state must be maintained in some manner within the module. However, there is no way to reach and test this fault state in normal operation. Since the network is not testable, it cannot be made self-checking. Q.E.D.

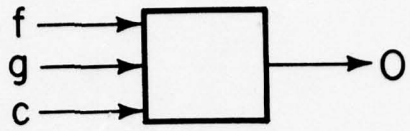


Figure 5.6a. Model of network (c,f,g,O)

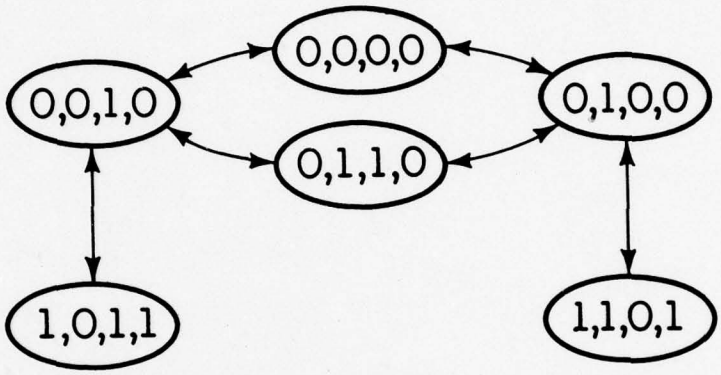


Figure 5.6b. Normal transition sequences

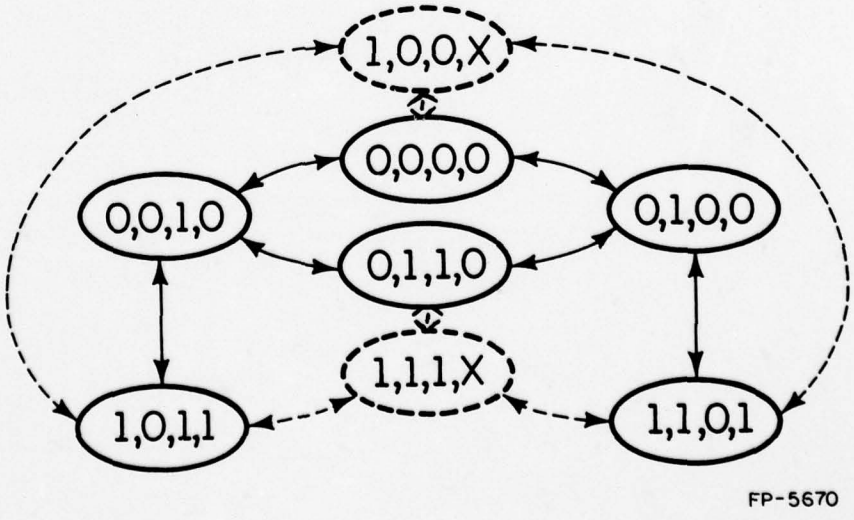


Figure 5.6c. Faulty transition sequences

If the module is assumed to not have a fault within it, then the internal state of (c,f,g) would never reach (1,1,1) in the analysis above. As was shown before, the connections to the module allow the system to be self-checking if the module is assumed fault-free.

Theorem 5.2 means that no completely self-checking system can be built with the normal logic gates available. There must be some nonstandard unit which is used as the hardcore for the system, or the module that is not self-checking must be replicated to obtain the desired reliability. This does not consider some other mode of operation such as transition oriented logic or pulse-mode logic which probably have other untestable faults.

One other possibility for providing self-checking capability in the system is to feed back the checker outputs so they may be latched during the next clock period, as shown in Figure 5.7. Once a faulty output is signalled by the checker it will then remain at that noncode word, (0,0) or (1,1). Presumably this status is displayed and the fault recognized by the operator. All operations since the last time the checker output was checked may be in error.

System-wide all the checkers in the system can be fed to one final checker without loss of the self-checking characteristic. Only this one checker needs to be monitored in assuring self-checking operation.

AD-A056 534 ILLINOIS UNIV AT URBANA-CHAMPAIGN COORDINATED SCIENCE LAB F/G 9/5
DESIGN OF DIGITAL SYSTEMS USING SELF-CHECKING ALTERNATING LOGIC--ETC(U)
OCT 77 S E WOODARD DAAB07-72-C-0259
UNCLASSIFIED R-788 NL

2 OF 2
AD A056534

END
DATE FILMED
9-78
DDC

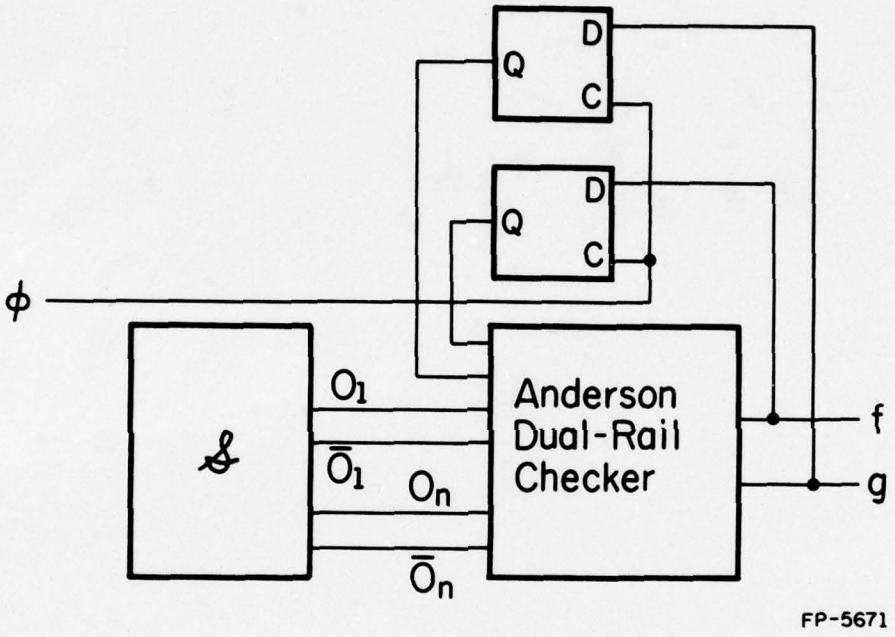


Figure 5.7. Feedback of checker outputs

6. SCAL MODULES IN NETWORK DESIGN

6.1. Introduction

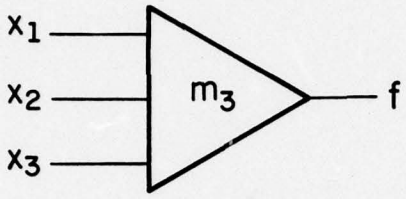
Recently a multi-valued logic gate was reported developed by industry [ELEC]. Through the years many such special types of gates have been studied both in theory and to a lesser extent in application. Among these special types of gates are the type known as threshold gates. And among threshold gates are gates known as majority gates and minority gates. These types of gates are particularly well suited for SCAL. The discussion here will be of the theoretical properties, with the realization that physical construction of them may be economical someday.

The minority module is represented as shown in Figure 6.1a. The truth table implemented is also shown. The majority module is similarly presented in Figure 6.1b. Two minority modules may be used to implement a majority module as shown in Figure 6.1c. The minority module has been shown to be a complete gate set by Reynolds [REYN1].

Theorem 6.1. The minority module is a complete gate set.

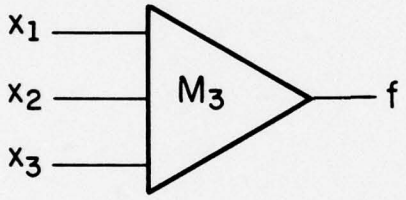
Proof: The 2-input NAND gate is well known to be a complete gate set [POST]. As shown in Figure 6.1d, a 2-input NAND gate can be implemented by the minority module. Therefore the minority module is a complete gate set. Q.E.D.

A further discussion about strong and weak completeness is given by Reynolds [REYN1]. The majority module is not a complete gate set since complementation cannot be done. Therefore, emphasis will be primarily on



f	$\bar{x}_1\bar{x}_2$	$\bar{x}_1x_2$	x_1x_2	$x_1\bar{x}_2$
$\bar{x}_3$	1	1		1
x_3	1			

Figure 6.1a. Minority module



	$\bar{x}_1\bar{x}_2$	$\bar{x}_1x_2$	x_1x_2	$x_1\bar{x}_2$
$\bar{x}_3$			1	
x_3		1	1	1

Figure 6.1b. Majority module

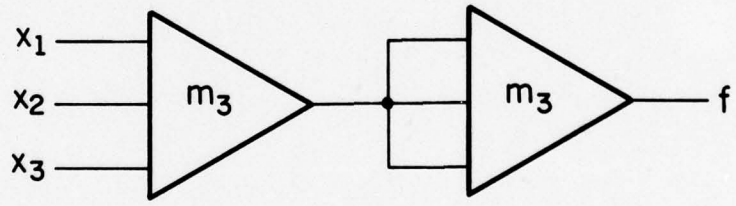
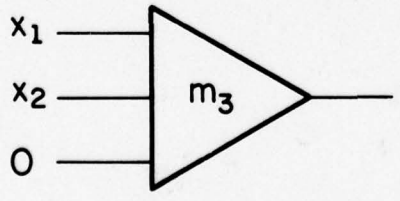


Figure 6.1c. Minority module implementation of majority module



FP-5672

Figure 6.1d. Minority module implementation of 2-input NAND

minority modules in this discussion.

For alternating logic, when the inputs alternate, the outputs must alternate. Assuming a period clock with successive values (0,1), a network composed of 2-input NANDS can be made an alternating network using minority modules. Let $m(X_1,X_2,\phi)$ represent the output of the minority module with inputs X_1,X_2, and period clock .

$$(F(X),F(\bar{X})) = (m(X_1,X_2,\phi),m(\bar{X}_1,\bar{X}_2,\phi))$$

$$= (m(X_1,X_2,0),m(\bar{X}_1,\bar{X}_2,1))$$

$$= (\overline{X_1X_2},\ X_1X_2)$$

$$= (\text{NAND}(X_1,X_2),\ \text{AND}\ (X_1,X_2))$$

Similarly a network of 2-input NOR gates can be made an alternating circuit by using the complement of the period clock (1,0).

$$(F(X),F(\bar{X})) = (m(X_1,X_2,\phi),m(X_1,X_2,\phi))$$

$$= (m(X_1,X_2,1),m(\bar{X}_1,\bar{X}_2,0))$$

$$= (\overline{X_1 \vee X_2},X_1 \vee X_2)$$

$$= (\text{NOR}(X_1,X_2),\ \text{OR}\ (X_1,X_2))$$

This provides the output value of the original network in the first time period. In the second time period the complemented value is output to provide the desired alternation. Since all lines in and out of the module alternate, by Theorem 3.6 the network is self-checking with respect to the gate.

6.2. Design With Minority Modules

Most networks require more than 2-input gates. For this reason it is necessary to consider whether a network with multiple input NAND gates can be directly transformed with minority modules into an alternating logic network.

In the analysis these symbolic definitions will be used:

A_L is an L input vector to a logic gate

$W(A_L)$ is the number of 1's in A_L

$m_I(A)$ is the minority function with an I input vecotr A

It is assumed that there is an odd number of inputs to the minority module. The minority function with input vector A is 1 if and only if $W(A_L) < L/2$. A NAND gate with input A is 1 if and only if $W(A_L) \leq (L-1)$ and an AND gate with input A is 1 if and only if $W(A_L) = L$. A few more symbol definitions are required:

$\overline{C}_k$ is an all 0 vector with k elements

C_k is an all 1 vector with k elements

$(\overline{C}_k, C_k)$ is a clock input, with k elements

$A \parallel B$ is the concatenation of two vectors

So $W(A \parallel B) = W(A) + W(B)$.

Theorem 6.2. For all NAND gates with N input vectors, X, there exist m_I such that for all $(X,\overline{X})$:

$$(m_I(X \parallel \overline{C}_K), m_I(\overline{X} \parallel C_K)) = (\text{NAND}(X), \text{AND}(X))$$

with K=N-1 and I=N+K=2N-1

Proof:

$$W(X \parallel \overline{C}_K) = W(X) + W(\overline{C}_K) = W(X)$$

$$m_I(X \parallel \overline{C}_K) = 1 \quad \text{iff} \quad W(X) < I/2$$

since $I = 2N - 1$ this is rewritten as

$W(X) < (2N-1)/2$ or $W(X) < N - 1/2$ or $W(X) \leq N - 1$

so $m_I(X \| \overline{C}_K) = 1$ iff $W(X) \leq N - 1$

$W(\overline{X} \| C_K) = W(\overline{X}) + W(C_K) = W(\overline{X}) + K$

$m_I(\overline{X} \| C_K) = 1$ iff $W(\overline{X}) + K < I/2$

this can be rewritten as $W(\overline{X}) < (2N - 1)/2 - K$

or $W(\overline{X}) < (N - 1/2) - (N-1)$ or $W(\overline{X}) < 1/2$ or $W(\overline{X}) = 0$

The conditions for NAND(X) and AND(X) are the same as those for the minority gate in the two time periods. Q.E.D.

Similarly a minority module implementation can be made from a NOR gate network.

Theorem 6.3. For all $(X,\overline{X})$,

$(m_I(X \| C_K), m_I(\overline{X} \| \overline{C}_K)) = (\text{NOR}(X), \text{OR}(X))$ with $K=N-1$ and $I=N+K=2*N-1$

Proof: The proof is similar to the proof of Theorem 6.2.

To clarify the application of the above theorems, an example will be given. A network of NAND gates and its truth table is given in Figure 6.2a. This is converted directly into minority modules using Theorem 6.2 as shown in Figure 6.2b. However, a more efficient implementation is shown in Figure 6.2c.

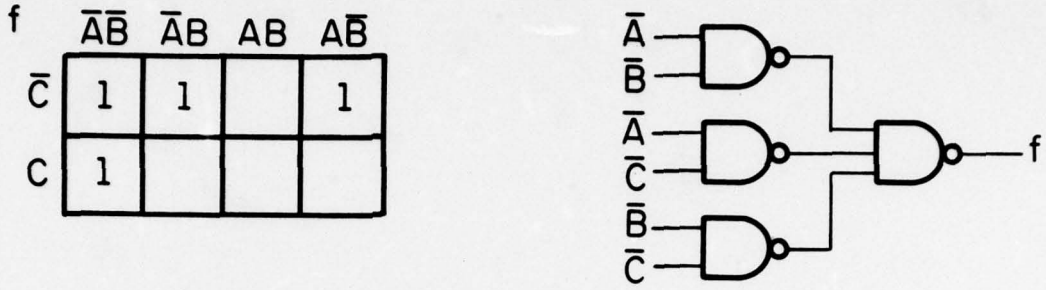


Figure 6.2a. Example network

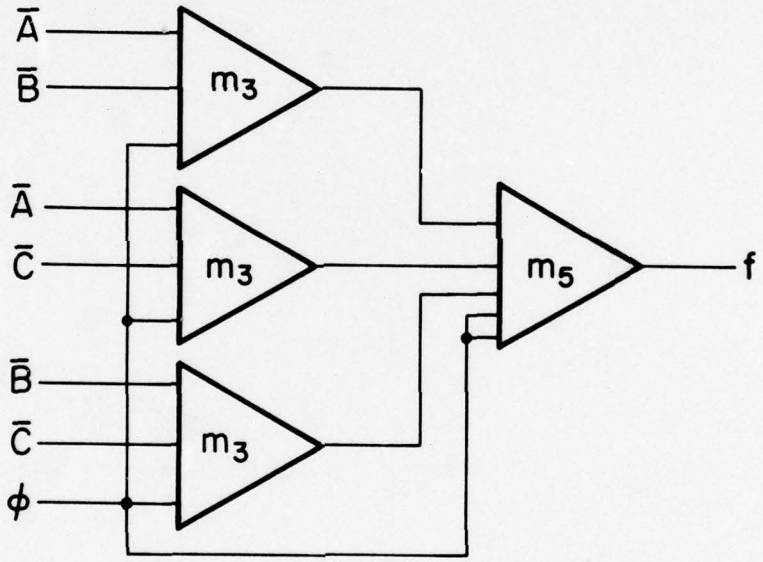
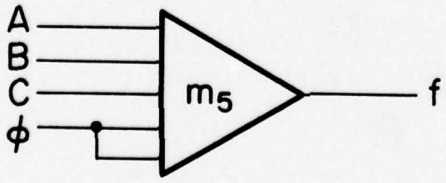


Figure 6.2b. Direct conversion to minority modules



FP-5673

Figure 6.2c. Minimal realization with minority modules

From the the contrived example it is clear that some functions may be very inexpensive to implement with minority modules. In this case the function to be implemented is a 3-input minority function. This is a self-dual function. When it is implemented with NAND gates, four NAND gates are required with nine total inputs. If each NAND is directly converted to minority logic by the method in Theorem 6.2 , then four minority modules are required with fourteen total inputs. However, a single minority module with three total inputs is all that is actually required. A technique to implement the function with minority modules directly from the truth table is useful. An appropriate weighting should be made for the cost of inputs also. The minority modules in general would have (N-1) clock inputs in the implementation of an n-input NAND gate. The design should try to minimize overall the number of such inputs. A branch and bound procedure such as that by Davidson [DAVI] would be appropriate. A detailed discussion of the implementation of functions using majority gates is given by Muroga [MURO]. The implementation with minority modules is similar and will not be presented here.

The minority modules of any size are self-checking if implemented in the manner presented in Theorems 6.2 and 6.3. This is because they all have alternating inputs and outputs for all input vectors. By Theorem 3.6 this means that each line is self-checking.

The implementation of a function using alternating logic and minority modules is a very straightforward approach to achieving self-checking. It relies upon the inexpensive availability of minority modules and a relatively small cost assigned to extra time required to achieve self-checking. In

instances where normal speed is desired and self-checking not required, the system could be easily switched to a nonalternating mode and could provide the desired operation.

7. SCAL COMPUTER DESIGN

7.1. Introduction

One of the primary applications for alternating logic is in computer systems where improved reliability is desired. In many instances the use of alternating logic could be the most cost-effective solution.

Section 7.2 will apply to computers some of the code translation design approaches discussed in Chapter 4. An analysis of the use of self-dual modules in computer system design will be done in Section 7.3. Finally, in Section 7.4 a discussion of SCAL use in large systems to achieve fault tolerance will be presented.

7.2. System Encoding Considerations

As was discussed in Chapter 4, it is desirable in system designs to match the code used to the failure mode of particular elements. For example, when all output lines of an element are independent and the cost of each additional output line is the same, a parity code is appropriate. A single-bit parity code is also appropriate for single line faults on busses or in the memory. However, in the central processing unit (CPU) generating a parity bit output is almost as costly as building an entire CPU. In this case an m-out-of-n code or Berger code is useful in space domain self-checking. Alternating logic has been shown to be effective as a time-domain approach. Therefore, the most cost-effective self-checking computer system should use a combination of codes dependent on the performance characteristics desired.

A model of a computer system is presented in Figure 7.1. Each element of the system has a greatly improved reliability if it is protected from undetected single faults. For many applications protection from single faults would be the most cost-effective mode of operation. An economic justification can be briefly given. Assume that functions exist which can describe: (1) a degree of fault protection in relation to the various faults protected against, (2) a measure of improved reliability for the system owner, (3) a cost for various designs which optimally achieve the reliability desired, and (4) a cost-benefit utility of reliability improvements (derived from functions 2 and 3). Typical plots of these last three functions relative to the first are shown in Figure 7.2. The functions have values only for certain discrete degrees of fault protection. For the types of costs and values shown in Figure 7.2, the peak utility is reached when single fault protection is used. The type of coding and design which can provide the minimum cost used in the graph can be analyzed. In a typical design process, a minimum cost for each type of fault protection desired would need to be determined before the utilities could be derived. In this case, assume the cost is known and it is desired to determine how it can be achieved.

The CPU could use alternating logic to provide the minimum hardware cost for single fault protection. The memory could use a parity code requiring only one bit per word. The translators discussed in Chapter 4 could be used by the CPU to communicate with the bus. A totally self-checking checker could be used to report to the outside world whether the system is operating properly. A code reply signal could be used with the peripheral devices on the input-output data paths. The reply signals would provide assurance that

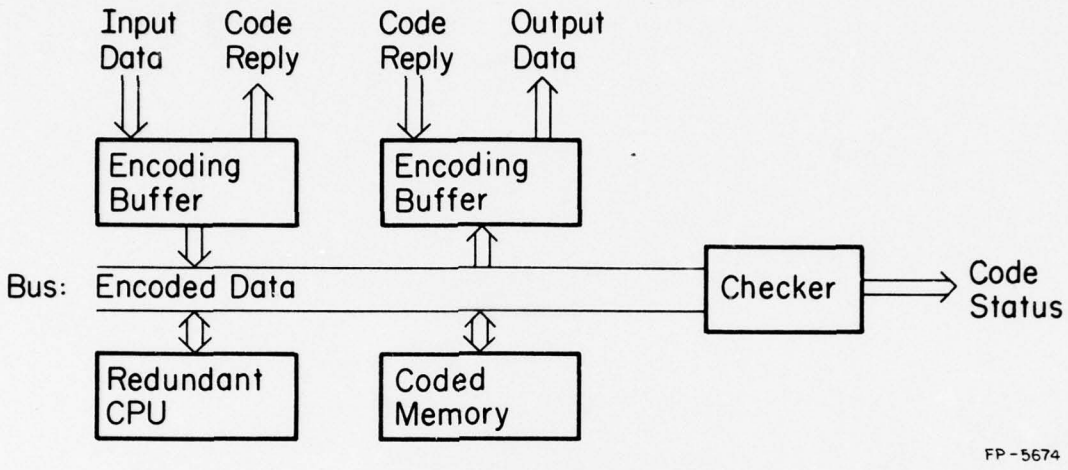


Figure 7.1. Computer system model

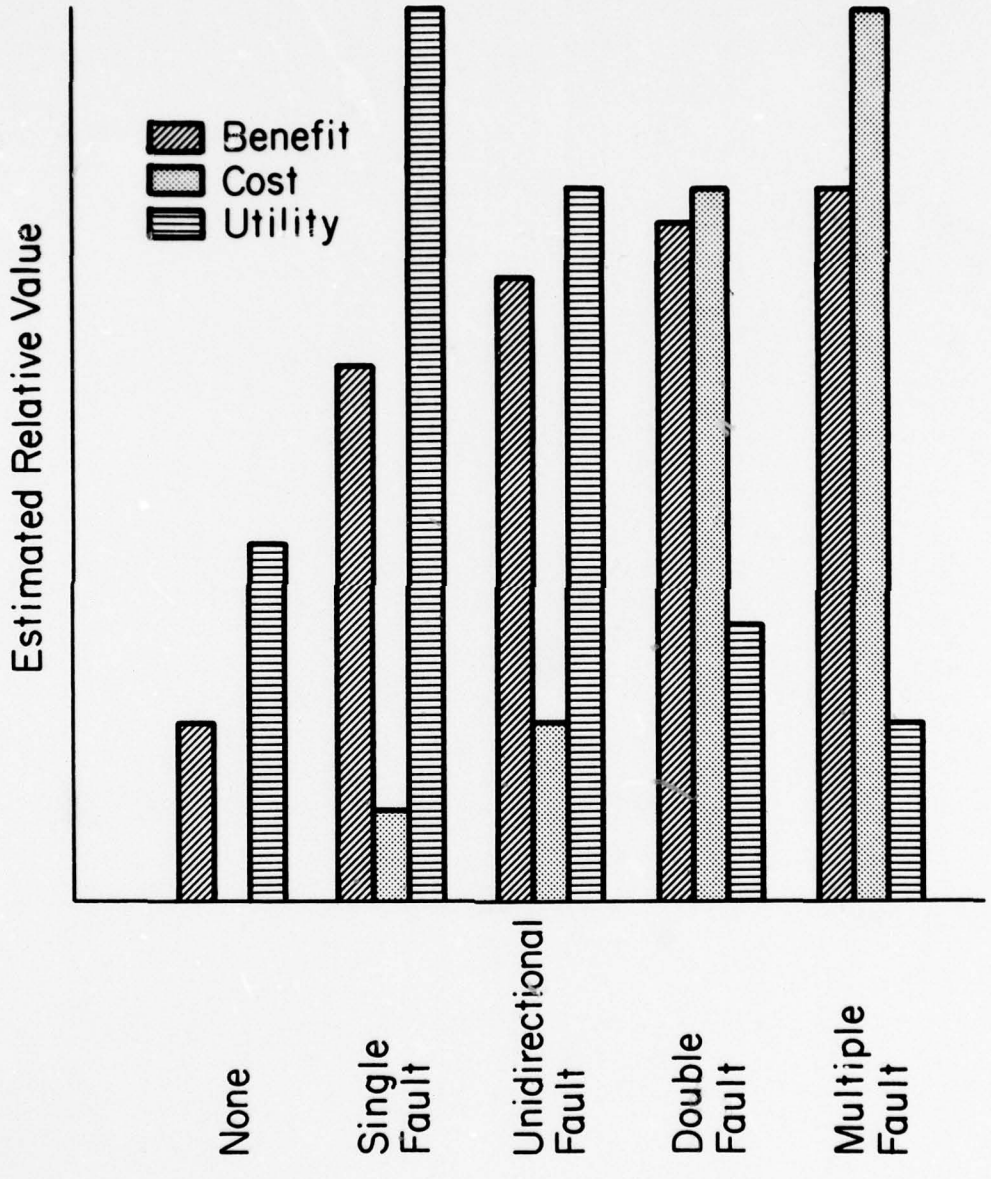


Figure 7.2. Reliability design trade-off

the correct data transfer had been made. Therefore the entire system would be protected from single faults. The resulting computer system is shown in Figure 7.3.

7.3. Self-Dual Modules

In designing the CPU it is often useful to combine certain function. Among these are arithmetic operations, the shift operation, and status determination. As was discussed earlier in Chapter 2, the adder is inherently self-dual. Similarly, the shift operation is self-dual. It can be easily implemented, as shown in Figure 7.4a, by using two flip-flops instead of the usual one. The status conditions can also be stored in two flip-flops as opposed to the usual one to achieve self-dual operation, as shown in Figure 7.4b. Alternatively, one of the encoding techniques for sequential machines could be used, as discussed in Chapter 4. By using these and other self-dual modules a SCAL CPU can be designed. The specific set of modules required would depend on the operations to be done by the CPU. The study of the design of an alternating logic CPU to replace a particular CPU would be useful in further research.

7.4. Large System Design With SCAL

Shedletsky [SHED2] has proposed the use of alternating logic in a system he refers to as an alternate data retry (ADR) system. The alternating capability of the system is utilized only upon occurrence of a fault. When the system detects a fault, the complemented signals are used and the correct values determined. This provides a method of achieving fault tolerance. It

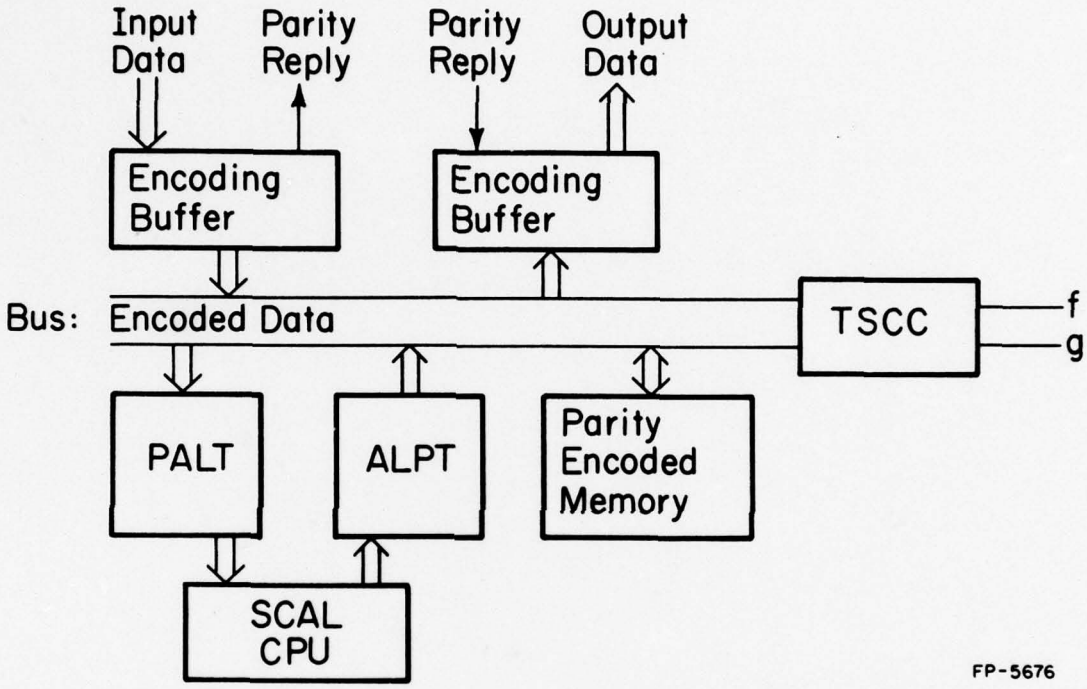


Figure 7.3. SCAL computer system

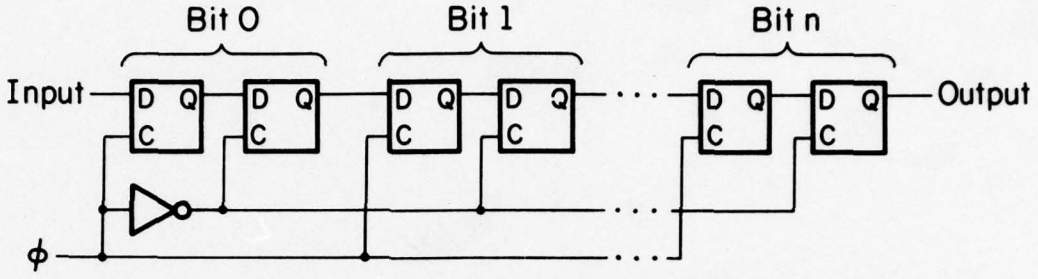


Figure 7.4a. Self-dual shift operation

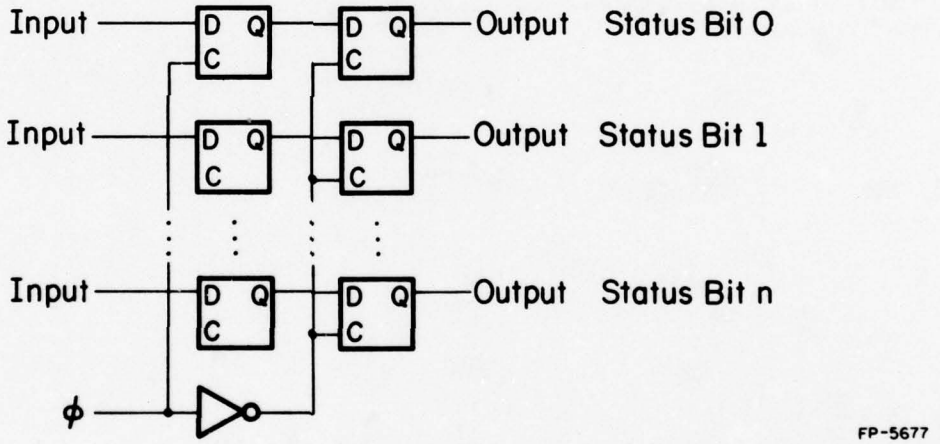


Figure 7.4b. Self-dual status storage

requires a check of the data during normal operation to determine whether the system is in error. To do this a space domain self-checking procedure was recommended. However, Shedletsky's approach requires double the amount of logic required in a self-checking system to achieve fault tolerance.

Let the cost of a normal system be N. Let the cost increase factor to convert a normal system to a space domain self-checking system be S and the cost increase factor to convert a given system to an alternating logic system be A. To convert a normal system to a space and time domain self-checking system, the cost is $A \cdot S \cdot N$, if the conversions are independently done. There is the possibility that a space domain self-checking system may be inherently self-dual, in which case the cost could be as low as $S \cdot N$. However, it is more likely that the space domain self-checking CPU is not self-dual. Assuming the space domain self-checking CPU is not self-dual, then A and S are approximately two. Thus the cost of building the ADR CPU is four times the normal CPU cost. This is probably worse than a triple modular redundant (TMR) CPU which has similar performance.

A SCAL CPU design which is more competitive with the TMR design is shown in Figure 7.5. In this case a normal CPU and a SCAL CPU are used in parallel to obtain space domain self-checking. In order to operate at the same speed as the normal CPU, the SCAL CPU uses only the first time period of the two time periods normally used in SCAL. When a fault is detected in either CPU, then the system operates at half-speed. The two time periods of operation of SCAL can be used with the normal CPU to generate three sets of output. A vote could be taken or the faulty member removed. This is comparable with TMR and may cost less than TMR if the value of A is less than two.

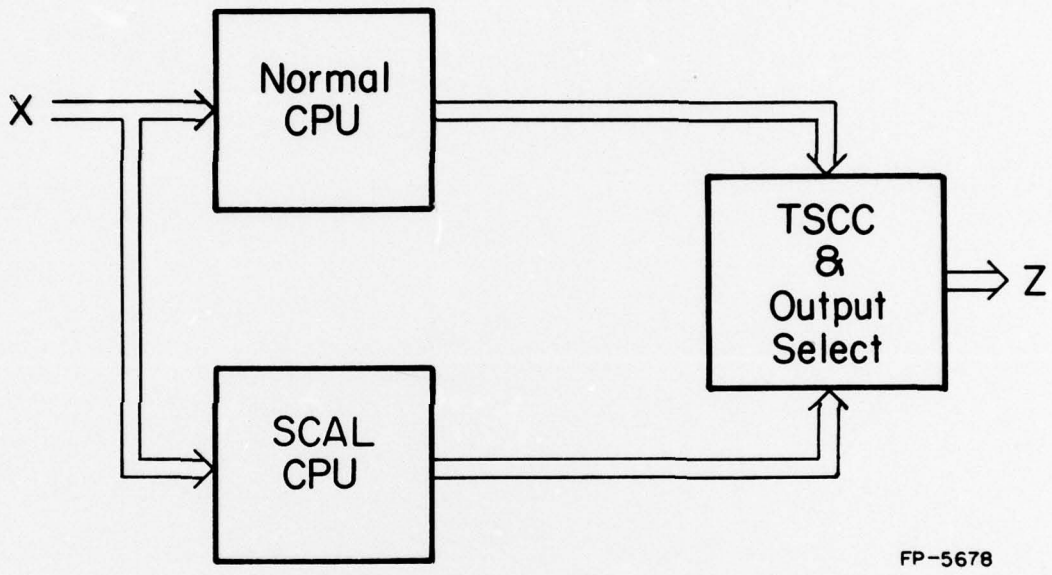


Figure 7.5. Fault tolerant alternating logic CPU

8. CONCLUSION

8.1. Summary

Analysis of the application of alternating logic to the design of self-checking systems has been done. In particular, results have been obtained in the areas of combinational logic, sequential machine design, SCAL checkers, SCAL modules, and SCAL system design. These will be briefly summarized.

Rules to evaluate whether a network is self-checking were presented. An alternating logic network is not self-checking if any line in the network does not satisfy at least one of the following criteria:

(1) It alternates for alternating inputs.

(2) It does not fanout and its path to the output is through unate gates.

(3) Path parity is the same for all paths from the line to the network output.

(4) It is an input to the same standard gate (NAND, NOR, AND, OR, or NOT) as an alternating line.

(5) Using the definition of line G and function F used in the thesis, it meets the condition

$$F(X,G(X)) \;\&\; (\overline{F}(X,0) \;\&\; F(\overline{X},0) \vee \overline{F}(X,1) \;\&\; F(\overline{X},1)) \neq 0$$

(6) If any line from a subnetwork used by more than one output fails to meet one of these conditions for an output, 1, then it is checked to see if:

$$\left[\sum_{k=1}^{n} \overline{F}_k(X,s) \;\&\; \overline{F}_k(X,s) \vee F_k(X,s) \;\&\; F_k(\overline{X},s)\right] \;\&\; F_1(X,G(X)) \;\&\; \overline{F}_1(X,s) \;\&\; F_1(\overline{X},s) = 0 \,.$$

A memory-efficient approach for the design of SCAL sequential machines was presented. In the design of large systems this approach is particularly valuable. A less practical approach to the design of SCAL sequential machines by direct implementation was also presented. This may sometimes have cost savings, but does not in general appear to be a desirable approach.

Two types of SCAL checkers were discussed. The implementation requirements for dependent input checkers were specified. Techniques of combining dependent and independent input checkers were presented. These achieve a minimal cost implementation of checking required for SCAL systems. In addition an analysis was given of the hardware requirements and the implications of alternative designs.

Minority modules were shown to be sufficient to implement in SCAL any NAND or NOR network. The code translation approach was shown applicable to system level design. In addition an effective use can be made of certain alternating logic modules in designing systems with improved reliability.

8.2. Current SCAL Research

The position of this thesis with respect to all the work done in this area is illustrated in Figure 8.1. From the figure it is evident that many of the aspects of logic design have been considered in their application to SCAL. Considerable work remains to be done.

I. Conceptual

Definition of Alternating Logic: Bark and Kinne [BARK]

Theory on Fault Detection of SCAL: Yamamoto, Watanabe, and Urano [YAMA]

Theory on Self-Checking: Anderson and Metze [ANDE1]

Theory on SCAL: Reynolds and Metze [REYN1]

Code Conversion in SCAL: this thesis

II. Combinational Logic

Initial Discussion: Bark and Kinne [BARK]

Two-Level Network's Fault Detection Properties: Yamamoto, et.al. [YAMA]

Costs of SCAL: Reynolds and Metze [REYN1]

Classes of Multiple Level Networks are SCAL: Reynolds and Metze [REYN1]

General Self-Checking Analysis: this thesis

Improve Design Techniques: for future research

III. Sequential Logic

Initial Discussion: Reynolds and Metze [REYN1]

Dual Flip-Flop Implementation: Reynolds and Metze [REYN1]

Code Conversion Technique: this thesis

Direct Implementation Approach: this thesis

IV. Self-Checking Checkers

Initial Discussion: Reynolds and Metze [REYN1]

General Designs: Reynolds and Metze [REYN1]

Dependent Line Checker Requirements: this thesis

Minimal Cost Checker Design: this thesis

Figure 8.1. Summary of current SCAL related work

Hardcore Logic Requirements: this thesis

V. Modules in SCAL

Initial Discussion: Ibuki, Naemura, and Nozaki [IBUK]

Uses in SCAL: Reynolds and Metze [REYN1]

Multiple Input Modules: this thesis

VI. Systems

Alternate Data Retry: Shedletsky [SHED]

System Design: this thesis

SCAL Functions: this thesis

Figure 8.1. Finished

8.3. Recommendations For Further Research

Some of the areas where future research may prove fruitful are evident from Figure 8.1. Specifically these include:

1. Constructive design procedures for combinational logic. The tools for analyzing whether a network is self-checking have been provided. It may now be possible to show techniques of designing SCAL.

2. Design techniques for sequential machines which do not require a checker on feedback variables. The complexity of this problem was presented in Chapter 4.

3. Further study of implementation of the direct conversion technique for SCAL sequential machines. This did not appear promising, but some new study may turn up worthwhile results.

4. More general application of hardcore analysis. The clock line was assumed to be the line to be used in controlling the system. A more general proof is probably possible. In addition, the application to space redundant self-checking systems should be straightforward.

5. Consideration of multiple faults in minority modules. Using additional redundancy of the minority modules, it should be possible to achieve self-checking for multiple faults.

6. System design using SCAL. Work similar to the space domain work of Ho and Metze [HO1,HO2] in the area of SCAL would be interesting.

In addition, work could be done on asynchronous implementation and different time encodings. However, the additional hardware cost does not present a promising opportunity for research in the use of different time

encodings.

One final note should be made of the merits of SCAL. This was discussed in detail in Chapter 2 and will be summarized. SCAL is not a universal solution to improving the reliability of digital systems. However, in applications where the cost of the additional time required for SCAL operation is not significant, SCAL can provide savings in hardware. Savings can also be realized in the physical pin count of the large scale integration devices in which SCAL is used.

REFERENCES

[ANDE1] Anderson, D., "Design of Self-Checking Digital Networks Using Coding Techniques," Coordinated Science Laboratory Report R-527, University of Illinois, September 1971.

[ANDE2] Anderson, D., and Metze, G., "Design of Totally Self-Checking Check Circuits for M-out-of-N Codes," IEEE Transactions on Computers, pp. 263-269, March 1973.

[BARK] Bark, A., and Kinne, C., "The Application of Pulse Position Modulation to Digital Computers," Proc. National Electronics Conference, pp. 656-664, September 1953.

[BREU] Breuer, M., and Friedman, A., Diagnosis & Reliable Design of Digital Systems, Computer Science Press, Inc., Woodland Hills, California, 1976.

[CART] Carter, W., and Schneider, P., "Design of Dynamically Checked Computers," IFIP 68, Volume 2, Edinburg, Scotland, pp. 878-873, August 1968.

[CHA] Cha, C., "Multiple Fault Diagnosis in Combinational Networks," Coordinated Science Laboratory Report R-650, University of Illinois, June 1974.

[DAVI] Davidson, E., "An Algorithm for NAND Decomposition of Combinational Switching Systems," Cooordinated Science Laboratory Report R-382, University of Illinois, May 1968.

[DUSS1] Dussault, J., and Metze, G., "A Low-Cost Totally Self-Checking Checker for 3N and Residue 3 Codes," Proc. of the 14th Annual Allerton Conference on Circuit and System Theory, Monticello, September 30-October 1, 1976.

[DUSS2] Dussault, J., "On the Design of Self-Checking Systems Under Various Fault Models," Coordinated Science Laboratory Report to be published.

[ELEC] "Four-Level Logic Coming Next Year from Signetics," Electronics, pp. 31-32, October 28, 1976.

[HO1] Ho, D., "The Study of a Totally Self-Checking Adder," Coordinated Science Laboratory Report R-582, University of Illinois, August 1972.

[HO2] Ho, D., "The Design of Totally Self-Checking Systems," Coordinated Science Laboratory Report R-723, University of Illinois, April 1976.

[IBUK] Ibuki, K., Naemura, K., and Nozaki, A., "General Theory of Complete Sets of Logical Functions," Electronics and Communications in Japan (IEEE Translation), Volume 46, Number 7, pp. 55-65, July 1963.

[KETE] Ketelsen, M., "An Integrated Circuit Fault Model for Digital Systems," Coordinated Science Laboratory Report R-743, University of Illinois, September 1976.

[KOHA] Kohavi, Z., Switching and Finite Automata Theory, McGraw-Hill, New York, 1970.

[LIU] Liu, T., Hohulin, K., Shiau, L., and Muroga, S., "Optimal One-Bit Full Adders with Different Types of Gates," IEEE Transactions on Computers, Volume C-23, Number 1, January 1974.

[MURO] Muroga, S., Threshold Logic and Its Applications, Wiley-Interscience, New York, Chapter 12, pp. 346-364, 1971.

[OZGU] Ozguner, F., "Design of Totally Self-Checking Asynchronous Sequential Machines," Coordinated Science Laboratory Report R-679, University of Illinois, May 1975.

[PITT] Pitt, D., "Design of Totally Self-Checking Asynchronous Sequential Machines," Report Number UIUCDCS-R-73-593, University of Illinois, September 1973.

[POST] Post, E., The Two-Valued Iterative Systems of Mathematical Logic, Princeton University Press, New Jersey, 1941.

[REYN1] Reynolds, D., and Metze, G., "Fault Detection Capabilities of Alternating Logic," Proc. 6th Annual Symposium on Fault Tolerant Computing, pp. 157-162, June 1976.

[REYN2] Reynolds, D., and Metze, G., "The Design of Alternating Logic Systems with Fault Detection Capabilities," Coordinated Science Laboratory Report R-738, University of Illinois, September 1976.

[REYN3] Reynolds, D., "Self-Checking Design Using Complete Sets of Alternating Primitives," Proc. of the 14th Annual Allerton Conference on Circuit and System Theory, Monticello, September 30-October 1, 1976.

[SHED1] Shedletsky, J., "A Rollback Interval For Networks with an Imperfect Self-Checking Property," Technical Report Number 96, Center For Reliable Computing, Stanford University, December 1975.

[SHED2] Shedletsky, J., "Error Correction by Alternate Data Retry," Technical Report Number 113, Center For Reliable Computing, Stanford University, May 1976.

[SMIT1] Smith, J., and Metze, G., "On the Existence of Combinational Networks with Arbitrary Multiple Redundancies," Proc. of the 13th Allerton Conference, October 1-3, 1975.

[SMIT2] Smith, J., "The Design of Totally Self-Checking Combinational Circuits," Coordinated Science Laboratory Report R-737, University of Illinois, August 1976.

[WAKE1] Wakerly, J., "Checked Binary Addition Using Parity Prediction and Checksum Codes," Technical Note Number 39, Digital Systems Laboratory, Stanford University, January 1974.

[WAKE2] Wakerly, J., "Partially Self-Checking Circuits and their Use in Performing Logical Operations," IEEE Transactions on Computers, Volume C-23, pp. 658-666, July 1974.

[YAMA] Yamamoto, H., Watanabe, T., and Urano, Y., "Alternating Logic and Its Application to Fault Detection," Proc. 1970 IEEE International Computing Group Conference, Washington, D.C., pp. 220-228, June 1970.

APPENDIX: ABBREVIATIONS, CONVENTIONS, AND GLOSSARY

A.1. Abbreviations

ALPT: Alternating Logic to Parity Translator

iff: If and only if

PALT: Parity to Alternating Logic Translator

SCAL: Self-Checking Alternating Logic

A.2. Glossary

Failure: Physical device malfunction

Fault: Logic representation of a failure

Function: Logic operation performed

Network: Implementation of a function

System: Combination of Networks

Translator: Network which transforms data from one code to another

A.3. Conventions

f: Fault

f^*: Self-dual function (Reynold's notation)

(f,g): Outputs of checker (Anderson's notation)

F: Function

F(X): Output from network implementing F when X is applied

$F(\overline{X})$: Output from network implementing F when $\overline{X}$ is applied

$\overline{F}(X)$: Complemented output from network implementing F when X is applied

$F(X,\overline{X})$: Alternating output from network implementing F when $(X,\overline{X})$ is applied

$F_f(X)$: Output from network implementing F when X is applied and fault f occurs

$F(X,G(X))$: Output from network implementing F, with value G(X) on line g, for input X

$F(X,s)$: Output from network implementing F, with value s on line g, for input X

F_i: Output of the i-th function

G(X): Value of line g in the network for input X

s: Logical value a faulty line is stuck-at (0 or 1)

X: Input vector

$(X,\overline{X})$: Alternating input sequence

$(Y,\overline{Y})$: Alternating sequence

$(\phi,\overline{\phi})$: Period clock alternating with time period of system, (0,1)

$(\overline{\phi},\phi)$: Complement of period clock, (1,0)

VITA

Scott Eugene Woodard was born in Urbana, Illinois on November 26, 1951. He has received a BS degree in 1973 and an MS degree in 1975 in Electrical Engineering from the University of Illinois, Urbana-Champaign. Currently he is in the Ph.D. program of the Electrical Engineering Department of the University of Illinois.